Implementation and Performance Evaluation of Black Hole Attacks on DSR and AODV in MANETs

Emmanuel Ahonsi Ailemen

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Eastern Mediterranean University
June 2016
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Cem Tanova
Acting Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Işık Aybay
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Assoc. Prof. Dr. Ali Hakan Ulusoy
Co-Supervisor

Asst. Prof. Dr. Gürcü Öz
Supervisor

Examining Committee

1. Assoc. Prof. Dr. Ahmet Rizaner

2. Assoc. Prof. Dr. Muhammed Salamah

3. Assoc. Prof. Dr. Ali Hakan Ulusoy

4. Asst. Prof. Dr. Yıltan Bitirim

5. Asst. Prof. Dr. Gürcü Öz

ABSTRACT

Mobile Ad-hoc Networks (MANETs) are type of wireless multi-hop networks that consists of group of mobile nodes that can interact with each other without the help of any predefined infrastructure or centralized controller such as a base station. MANETs are easily prone to security attacks. Black hole attack is one of such attacks. In a black hole attack, a malicious node deceitfully publicize itself as having the shortest path to a given destination and drops the data packet without forwarding it to the actual destination. Dynamic Source Routing (DSR) and Ad-hoc On-demand Distance Vector (AODV) protocols are two well-known protocols used for routing in the MANETs. In this thesis, we investigate the effect of black hole attack in both DSR-based and AODV-based MANETs. For this purpose, we implement both protocols using Network Simulator version 2 (NS-2) to obtain the performance in terms of Packet Delivery Ratio (PDR), End-to-End Delay (EED) and throughput with black hole attack and without black hole attack. A modified approach is implemented for DSR and AODV protocols to improve their performance in the presence of black hole attacks. Simulation results obtained show that network performance is reduced for both protocols in the presence of balckhole attack and AODV protocol is more affected by the black hole than DSR protocol. The results of the Modified schemes obtained show a significant improvement in the network performance over DSR and AODV protocols with black hole.

Keywords: MANET, DSR, AODV, black hole, PDR, EED, throughput.

ÖZ

Gezgin alt yapısız ağlar (MANETs) bir grup gezgin cihazdan oluşan ve herhangi bir alt yapıya veya merkezi bir erişim noktasına ihtitaç duymadan haberleşmeyi olanaklı kılan kablosuz ağ çeşitlerindendir. MANET'ler ağ güvenliği saldırılarına açık olup, kara delik saldırıları güvenliği tehdit eden başlıca saldırı türlerinden bir tanesidir. Kara delik saldırıları, kötü niyetli bir düğümün belirli bir hedefe ulaşmak için en kısa yolun kendi üzerinden geçerek ulaşılabileceği şeklinde yanlış bir bilginin ağdaki tüm düğümlere bildirmesiyle oluşur ve hedefe ulaşmak için gönderilmiş olan veri paketleri kara delik tarafından hedefe gönderilmeden imha edilir. Dynamic Source Routing (DSR) ve Ad-hoc On-demand Distance Vector (AODV) protokolleri MANET'lerde kullanılan en bilinen yönlendirme prokolleridir. Bu çalışmada, DSR ve AODV protokolleri kullanılarak MANET üzerindeki kara delik saldırılarının etkisi incelenmiştir. Kara delik etikisini inceleyebilmek için iki protokol kara delik bulunan ve bulunmayan durumlarda Ağ Simülatörü NS-2 de uygulanmış ve Paket Teslim Oranları (PDR), Noktalar Arası Gecikme (EED) ve verim sonuçları incelenmiştir. Kara delik saldırısının DSR ve AODV protokolleri üzerindeki etkisini azaltmak üzere bu protokollerde yeni bir düzenleme yapılmıştır. Benzetim çalışmalarından elde edilmiş sonuçlar göstermiştir ki kara delik saldırıları ağın performansını her iki protokol için de düşürmekle birlikte AODV protokolü kara delik saldırılarından DSR protokolüne göre daha fazla etkilenmektedir. Ayrıca benzetim sonuçları, yeniden uyarlanmış olan DSR ve AODV protokollerinin başarım sonuçlarının kara delik saldırıları karşısında yükseldiğini göstermiştir.

Anahtar kelimeler: MANET, DSR, AODV, kara delik, PDR, EED, verim.

DEDICATION

Dedicated to God Almighty and to My Family
For Their Love and Support

ACKNOWLEDGEMENT

First and foremost, I acknowledge and appreciate God almighty for grace upon my life and his abiding presence with me throughout my stay and study in this school.

Special thanks and appreciation goes to my supervisors Assoc. Prof. Dr. Ali Hakan Ulusoy and Asst. Prof. Dr. Gürcü Öz for their collaborative and relentless effort during the course of this thesis to ensure the success of this work. Their guidance on every step during this research has been an immerse help to me and it is great privilege to work with them.

I would love to use this opportunity to appreciate my wonderful parents Mr. and Mrs. Eguaoje A.I. Nicholas for their love, prayers and support. They have been my financial backbone during the course of my study. I own my achievement to them.

Sincere thanks to my siblings: Ailemen Ohis and Ailemen Ruth. To my beloved Igbinosa Emwinghare, my extended family: Mr. and Mrs. Eguaoje Ajayi and family, Mr. and Mrs. Asije and family, Mr. Isaac and family, Mr Andrew and family, Godspower Igbora, Abraham and lots of others.

Thanks to all the departmental lecturers, my friends: Oyedeji Ajibola, Olaifa Femi, Flora, Banke, Ayoku Temitope, Zuhir and others. Special thanks to all scripture miners unit members, deeper life members North Cyprus and Bethesda church. Love you all and God bless you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATION

AODV	Ad-hoc On-demand Distance Vector
AWK	Alfred Weinberger Kernighan
Bps	Bits per second
CBR	Constant Bit Rate
DoS	Denial of Service
DSDV	Destination Sequenced Distance Vector
DSR	Dynamic Source Routing
EED	End-to-End Delay
FSR	Fisheye State Routing
IP	Internet Protocol
MANET	Mobile Ad-hoc Network
ms	Milliseconds
NAM	Network Animator
NS-2	Network Simulator version 2
OLSR	Optimized Link State Routing
OSPF	Open Shortest Path First
OTCL	Object Oriented Tool Command Language
PDA	Personal Digital Assistant
PDR	Packet Delivery Ratio
RERR	Route Error
RREP	Route Reply
RREQ	Route Request
RREQ ID	Route Request Identification
TCL	Tool Command Language

TTL	Time To Live
UDP	User Datagram Protocol
ZHLS	Zone-based Hierarchical Link State
ZRP	Zone Routing Protocol

Chapter 1

INTRODUCTION

1.1 General Overview

A group of mobile nodes that are wirelessly connected together and can communicate with each other via radio waves is known as a wireless network. A wireless network can be categorized into two classes. The class of wireless network that operates with the support of a fixed infrastructure such as an access point is known as infrastructure-based network and the wireless network that does not require the support of a fixed infrastructure is referred to as infrastructure-less network. Mobile Ad-hoc Networks (MANETs) fall into the class of infrastructure-less network. In MANETs, each node has the ability to join or leave the network at any given time because of the networks open nature and can act not only as a host but also as a router to help in forwarding packet if it is neither the source nor destination using a routing protocol. For routing in MANETs, routing protocols such as Destination Sequence Distance Vector (DSDV), Optimized Link State Routing (OLSR), Dynamic Source Routing (DSR) and Ad-hoc On-demand Distance Vector (AODV) have been developed for efficiently transmitting packets from source to destination.

1.2 Problem Statement

In MANETs, due to its characteristic and nature such as open communication medium, lack of centralized monitoring infrastructure and node cooperativeness cause the network to be easily susceptible to various types of security attacks. Some of these attacks include Denial of Service (DoS), black hole attacks, gray hole attacks, worm

hole attacks, etc. [1, 2]. Routing protocols were created without taking into consideration the security issues prone to MANETs. Therefore, MANETs using these protocols are usually susceptible to these security attacks.

1.3 Motivation

Security is highly essential for both wireless and wired network for the safeguarding of data communication. Therefore, the security in MANETs is one of the major issues for the standard functioning of the network. The level of trust of an individual in a network is based on its level of security which should satisfy availability, confidentiality, integrity, authentication and authorization [2]. MANETs are easily prone to security attacks and black hole attack is one of the most common type of attack faced by this network where a malicious node drops data packets thereby causing harm to the network. We therefore, focus on black hole attack because of its common nature and the high adverse effect it has on the network.

1.4 Thesis Objectives

This thesis focuses on analyzing the effect of black hole attack on the network performance of both DSR and AODV protocols for MANETs and aims at improving the original existing DSR and AODV routing protocols in the presence of a black hole attack. In this thesis, a modified scheme is implemented for both DSR and AODV routing protocols to detect and mitigate the effect of the black hole attack and thereby improve the performance of the network in the presence of the malicious node. This thesis also aims at comparing and analyzing the results of the modified DSR and AODV routing protocols with that of the original DSR and AODV routing protocols using throughput, Packet Delivery Ratio (PDR) and End-to-End Delay (EED) as the performance metric to evaluate the performance of the network.

1.5 Thesis Structure

The remaining part of this thesis are divided into six chapters.

Chapter 2 deals on the literature review and background studies on MANET, DSR and AODV routing protocols that includes discussions of black hole attack on these routing protocols.

Chapter 3 presents the detailed methodologies of modified scheme for both DSR and AODV protocols together with the performance metrics used for the simulations.

Chapter 4 presents a detailed description of the network environment used and the results obtained from the simulations showing the PDR, throughput, and average EED.

Finally, Chapter 5 talks about the conclusion and presents the future works to be done as a continuation of this topic.

Chapter 2

LITERATURE REVIEW

2.1 Mobile Ad-Hoc Networks

MANETs are a kind of network that consist of a collection or groups of self-organizing wireless mobile hosts that can communicate with each other directly without the aid of a centralized supporting infrastructure such as access points, routers or base-stations. Each device on this network communicates directly with another device wirelessly if they are on reachable communication range. Otherwise communication is done via multi-hop communication which connectivity is done by forwarding packets to neighboring devices so as to get a route to the destination device [2]. In MANETs, each mobile device behaves not only as an end user but also as a router to other devices on the network. Dynamic network topologies are formed by MANET since each node can leave or join the network freely [1, 2]. Examples of these communicating mobile devices include handheld digital devices, phones, Personal Digital Assistants (PDAs), laptops, etc. Figure 2.1 gives a typical example of a MANET. Data communication is done with the help of a routing protocol.

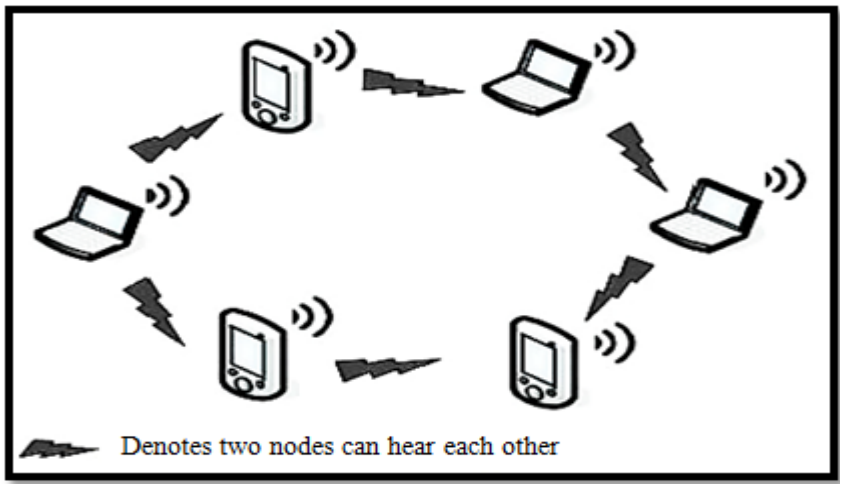

Figure 2.1: A typical MANET

2.2 Routing Protocols in MANETs

Routing is defined as a process of finding and selecting the best path among other paths in a network on which to forward data packets between a source host and a destination host.

2.2.1 Classes of Routing Protocols in MANETs

As shown in Figure 2.2, MANET routing protocols are sub-divided into three classes which are based on the way they function. They are proactive protocols, reactive protocols, and hybrid protocols [3].

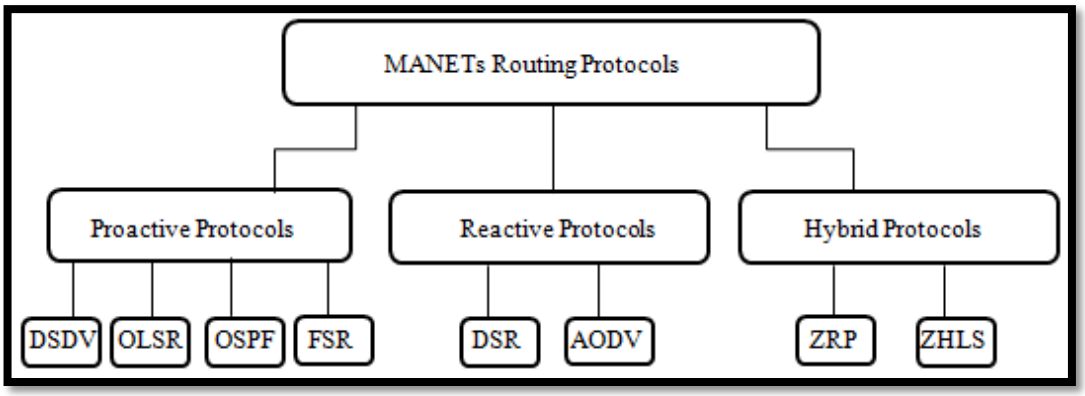

Figure 2.2: Classes of MANET routing protocols

Proactive Protocols: These protocols are also known as table driven protocols [3]. For these protocols, each node in the network maintains routing tables for routing

information in the network. These tables in each node are continuously updated whenever there is a change in the network topology through the propagation of these changes and each node in the network has the complete routing information of the entire topology [3]. Examples of proactive protocols include DSDV, OLSR, Open Shortest Path First (OSPF), and Fisheye State Routing (FSR) protocols.

Reactive Protocols: These protocols are also known as on-demand protocols because they do not maintain routing information on the nodes in the absence of communication but only search for a route when a node needs to send a packet to another node in the network [4]. The path discovery is done by broadcasting of Route Request (RREQ) packets throughout the network until RREQ is received by the destination node. Examples of reactive protocols include DSR, and AODV.

Hybrid Protocols: These protocols combine both the proactive and the reactive approaches in its mode of operation to achieve a better result. Hybrid protocols are able to minimize the issues of delay which can be seen in reactive protocols and control overhead found in proactive protocols. Examples of hybrid protocols include Zone Routing Protocol (ZRP) and Zone-based Hierarchical Link State (ZHLS) routing protocol [4].

2.3 Dynamic Source Routing

DSR is a reactive or on-demand source routing protocol [5]. This implies that a source only initiates route discovery if it has a packet to send and then places the complete routing information to the destination in the data packet header. Each node maintains a route cache which contains routes the node has knowledge of. A node may cache multiple routes to a single destination such that if one fails, it falls back on the other. When a node decides to send a packet to a destination, it searches its route cache to

check if it has a route to the given destination and if there exists a route to this destination it includes the routing information into the data packet and uses the route to forward the packet to the destination node. In the absence of a route to the destination node, the source node initiates a route discovery process to obtain routes to the destination. Also, in a situation of link failure or route breakage, a mechanism known as route maintenance is initiated. Therefore, the two mechanisms used in the DSR are route discovery and route maintenance [5, 6]. The control packets used during route discovery are RREQ, Route Reply (RREP) and Route Error (RERR) packets.

2.3.1 Route Request Packet

A source in a network that wants to initiate communication with a different node needs to transmit a RREQ packet. The RREQ packet as shown in Figure 2.3 includes source address, destination address, and a unique identification number known as (RREQ ID) and path [6]. There is also a hop limit which is carried out using the Time To Live (TTL) value in the RREQ packet. The TTL value shows the number of hops that the RREQ packet should be forwarded and each intermediate node receiving the RREQ packets decrements the hop count by one before forwarding.

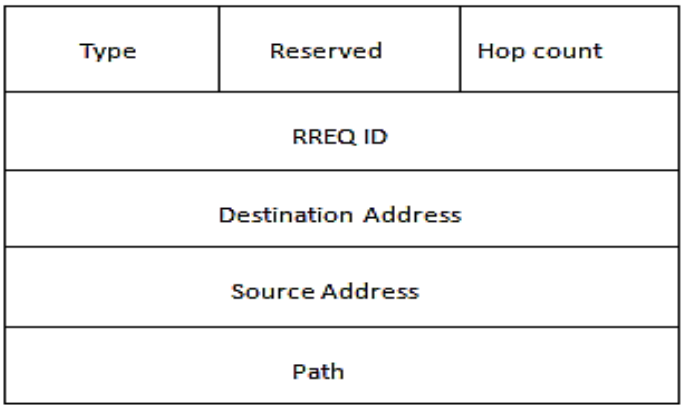

Figure 2.3: RREQ packet structure in DSR

2.3.2 Route Reply Packet

The destination node or any intermediate node in the network that has a route to the destination generates a RREP packet and forwards it back to the source node. The RREP packet as shown in Figure 2.4 mainly includes destination address, source address, life time and path.

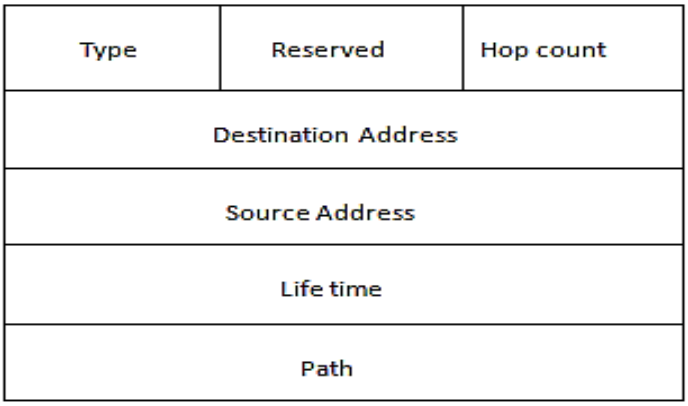

Figure 2.4: RREP packet structure in DSR

2.3.3 Route Error Packet

During data transmission, if a link to a next hop neighbor is broken, a RERR packet is generated and propagated backward to the source. The RERR packet contains information on the broken link which is used by the source node to identify and erase any route in its cache that contains the failed link.

2.4 DSR Route Discovery Mechanism

The following steps below give a detailed procedure for the operation of route discovery in DSR when no route exists in a route cache of the source node intending to forward a packet to a destination node.

1. The source node broadcasts a route RREQ packet to all its neighbors. The RREQ packet contains the source address, destination address, and a unique RREQ ID which is selected by the source node.

2. When an intermediate node receives the broadcast RREQ packet, it checks its cache for a route to the destination. If it has no route to the destination, it appends its address into the path field and broadcasts the RREQ packet to its neighbors as shown in Figure 2.5. A node discards a RREQ packet if it has recently seen another RREQ packet from the same source with the same RREQ ID and destination address or if its own address is among the Internet Protocol (IP) addresses listed in the route record of the RREQ packet.

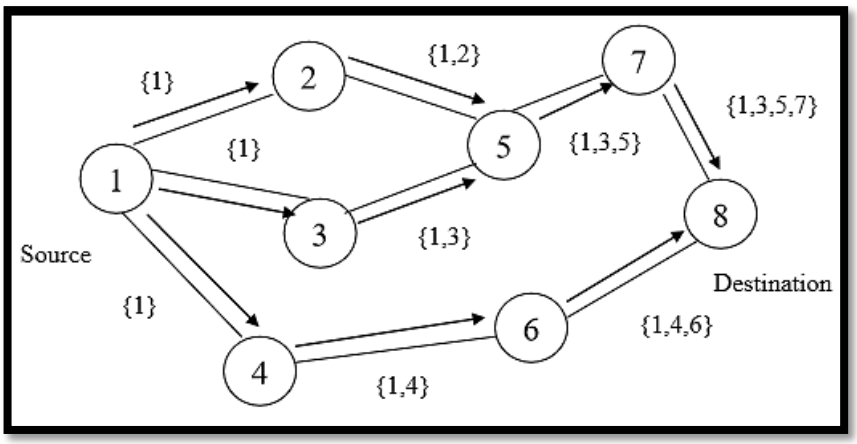

Figure 2.5: A route discovery operation in DSR

If an intermediate node has a route to the destination, it generates and returns a RREP packet to the source with this RREP packet containing the cached route information to the destination attached with the accumulated route record copied from the RREQ packet. But if the RREP packet is generated by the destination, it places the route record in the RREQ packet in the RREP packet and forwards it as a unicast packet to the source as seen in Figure 2.6. This reverse-route forwarding of the RREP packet by the destination is only obtainable if network links are bi-directional [6]. The following steps below give the procedure for the source after receiving the RREP packets in DSR.

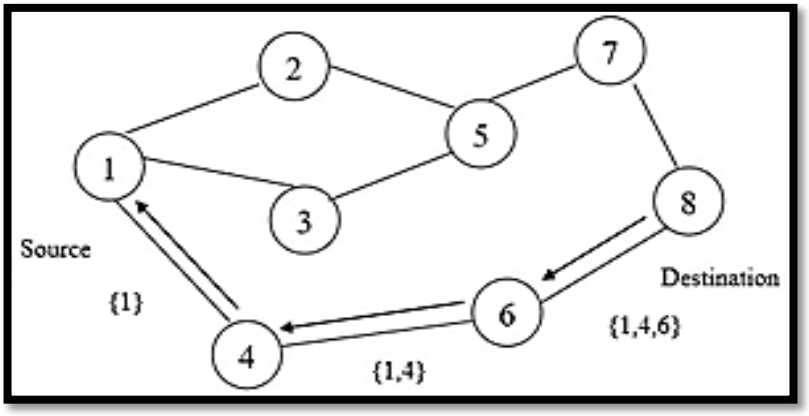

Figure 2.6: A route reply operation in DSR

Step 1. The source node on receiving the RREP packets, extracts and stores the source routes obtained from the RREP packets into its route cache for routing of data packets to the given destination.

Step 2. Source node selects from its route cache the route with the shortest path to the destination and begins the transmission of data packet through it.

2.5 DSR Route Maintenance

Route maintenance in DSR protocol is achieved using passive acknowledgments [6] and RERR packets. In a network topology, when there is a link failure, the source node is informed by the use of RERR packet. Each node in an active route uses passive acknowledgment to confirm that the next hop neighbor receives the packet by listening if the neighbor receives the packet or not. If a node cannot confirm that the packet is received by the neighbor node, the link is considered broken and an RERR packet is sent back to the source node which removes the hop in error from its cache and also all the routes that contain this hop must be truncated at that point. The source node then uses another existing route in its cache that does not have the broken link to forward the data packet to the destination otherwise restarts a new route discovery.

Using an illustration as in Figure 2.7, a route is discovered and established from source node A to destination node E as A-B-C-D-E. Node A routes the data packet through intermediate nodes B, C and D. Considering the link between node C and D fails, node C sends a RERR packet to node A along route C-B-A. Nodes C, B, A on receiving the RERR packet update their route caches by removing link C-D.

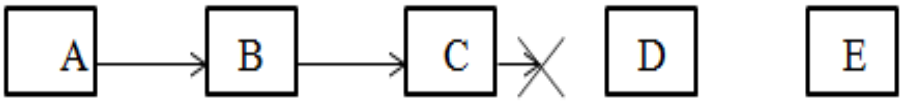

Figure 2.7: Node C is not able to forward the packet through node D

2.6 Black Hole Attack in DSR

In DSR when a source node has a data packet to send to a destination node in the network and does not have a route in its cache to the destination, it initiates a route discovery process to find a route to the destination. It is during the route discovery process that a malicious node is able to advertise itself as having the shortest route to a destination [7, 8]. The steps below show the process of black hole attack in DSR.

1. The malicious node waits to receive a RREQ packet sent by a source node on the network.

2. The malicious node receiving the RREQ packet, it immediately generates and sends a falsified RREP packet to the source node without checking its route cache as having the shortest path to the destination node [9].

3. The falsified RREP packet is sent as a unicast packet to the source node directly or through its immediate neighbors who then forward the message towards the source node.

4. The source node that receives the falsified RREP packet from the malicious node updates its routing cache and thinks the route discovery process is ended. It disregards the RREP packets from other nodes in the network and begins to transmit the data packets through the malicious node.

5. The malicious node that receives the data packets drops them without forwarding them towards the destination node creating a black hole in the network as shown in Figure 2.8.

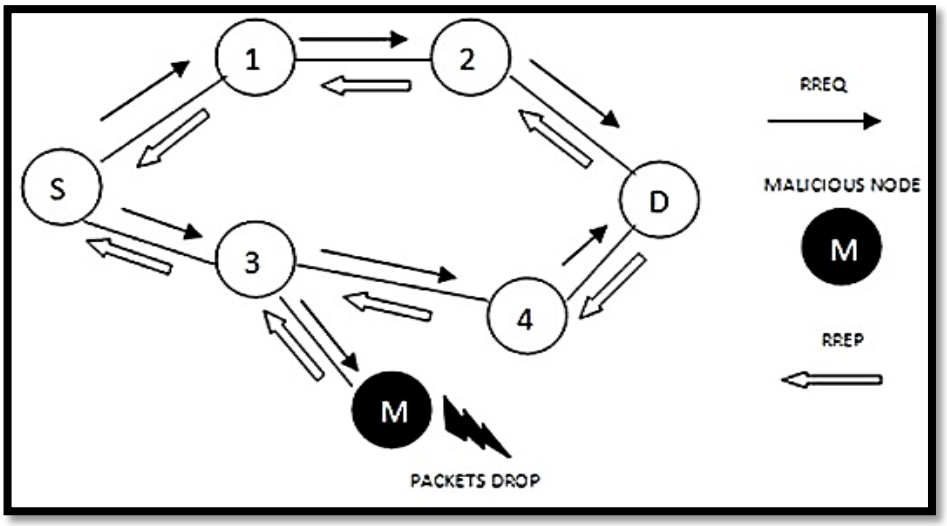

Figure 2.8: An illustration of a black hole attack in DSR

2.7 Ad-hoc On-demand Distance Vector

AODV is a reactive or on-demand protocol. This protocol is an improved version of the DSDV protocol [10]. AODV is invoked only when a source node in a network has a data to transmit hence the name on-demand. Each node in the ad hoc network acts as both a router and a host. All nodes maintains a routing table that contains information about known destinations which are used to route data packets to the desired destination. The routing table is minimized to only include the next hop information and not the entire route information to the destination node [10]. Sequence number is a distinct feature used by AODV when comparing it to other routing

protocols. Sequence number increases monotonically whenever a RREQ packet is sent or a RREP packet is forwarded in response to a RREQ packet received by a destination node. It is used to indicate the freshness of a routing information and for loop prevention. The higher the sequence number of a destination in a routing information is, the fresher (more recent) the route is [10]. When a node has a data packet to send, it checks its own routing table for the next hop information to the destination. If such a next hop entry exists, then the data packet is sent through it else a route discovery process is initiated. In AODV functionality, the basic control packet set comprises of RREQ, RREP, RERR and Hello packets. Description of these packets is given below.

2.7.1 Route Request Packet

A RREQ packet is generated by a source node and forwarded via flooding when the source node needs to discover a route to a given destination node for data transmission. The RREQ packet consists of RREQ ID, source IP address, destination IP address, source sequence number, destination sequence number and hop count [10]. As the RREQ packet propagates in the network, intermediate nodes use it to set up reverse route entry toward the direction of the source node in its routing table. The fields of RREQ packet as shown in Figure 2.9 are discussed below:

- Type: This indicates the packet type which is 1 representing RREQ packet.
- Hop Count: This is the number of hops from the source node to the node currently handling the RREQ packet.
- RREQ ID: This is a numeric value attached to a particular RREQ packet used to uniquely identify the RREQ packet in connection with the IP address of the source node. It assists other nodes in a network to identify and discard duplicate RREQ packets which they receive. RREQ ID value is incremented whenever a new RREQ packet is broadcasted by the source node.

- Destination IP Address: This is the IP address of the destination node that a route is desired for.
- Destination Sequence Number: This is the most recent sequence number received previously by the source node for any given route for the destination node.
- Source IP Address: This is the IP address of the source node that originates the RREQ packet.
- Source Sequence Number: This is the current sequence number of the source node that would be used during routing.

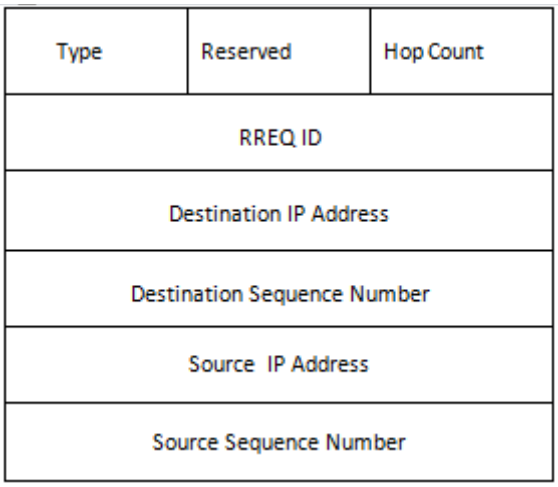

Figure 2.9: RREQ packet structure in AODV

2.7.2 Route Reply Packet

A RREP packet is created if a node finds out a route to a RREQ packet it received. The RREP packet is unicasted to the source node since AODV supports bi-directional links. RREP packet as shown in Figure 2.10 consists of fields such as type, hop count, destination IP address, destination sequence number, source IP address and lifetime. These fields have been discussed in the section 2.7.1 except for lifetime. Lifetime is the time duration for which a node that receives the RREP packet considers the route to be unexpired. If the RREP packet is generated by the destination node, the hop count will

be equal to zero but if the RREP is being generated by an intermediate node the hop count will be its distance to the destination node.

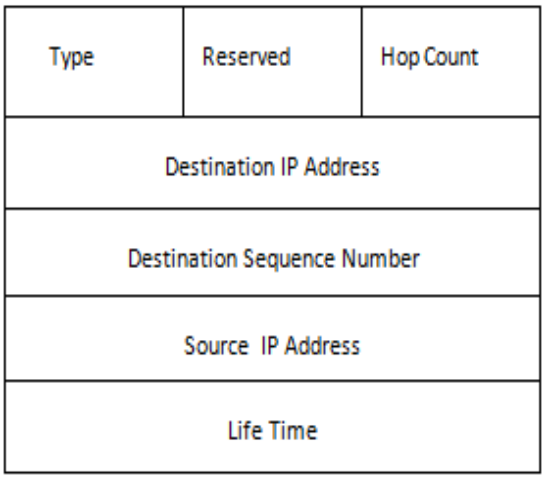

Figure 2.10. RREP packet structure in AODV

2.7.3 Hello and Route Error Packets

Hello packets are used to detect the link failures. During transmission of data between source and destination nodes, neighboring nodes which are part of the active route periodically send *Hello* packets to each other to confirm connectivity. The absence of *Hello* packet indicates the link failure which causes RERR packet to be generated and broadcasted to neighbors with the aim of notifying the source and other nodes in the network about the broken link.

2.8 Routing in AODV

Routing in AODV takes place in two phases. First is the route discovery phase which commences when a source node having a data packet to send does not have the next hop route entry for the destination node. Second is the route maintenance phase which takes place during data transmission [10]. The following steps below give the description of AODV routing with Step 1 to Step 4 dealing with the route discovery phase and Step 5 on route maintenance phase.

Step 1: The source node generates a RREQ packet and broadcasts it via flooding to all neighboring nodes in order to find a route to the destination.

Step 2: Intermediate nodes receive the RREQ packet and do the following actions:

- Discard the RREQ packet if they have been previously seen and processed. But if it is not a duplicated RREQ packet, they generate and forward a RREP packet to the source, if a fresh route exists for the destination node. By fresh it is meant that that the sequence number of the destination node stored in the routing table is greater or equal to the sequence number of the destination node in the RREQ packet just received.
- If no fresh route exists to the destination, they extract the route information from the RREQ packet and make a reverse route entry in the routing table [10]. They then increment the hop count and rebroadcast the RREQ packet to neighbors until the destination is reached.

Figure 2.11 shows this process. The source as node 1 seeking a route to destination node 8 broadcasts the RREQ packet to its neighboring nodes 2, 3 and 4. Neither node 2, 3 nor 4 know a route to node 8, so each node creates a reverse route entry for the source and rebroadcasts the RREQ packet. We assume that the broadcast from node 2 arrives at node 5 before node 3 therefore, node 5 processes the broadcast from node 2 and discards that of node 3 as duplicate. After nodes 5, 6 and 7 broadcast, the RREQ packet finally arrives at the destination node 8. The first arriving RREQ packet to node 8 will be processed and other subsequent RREQ packets will be dropped as duplicates.

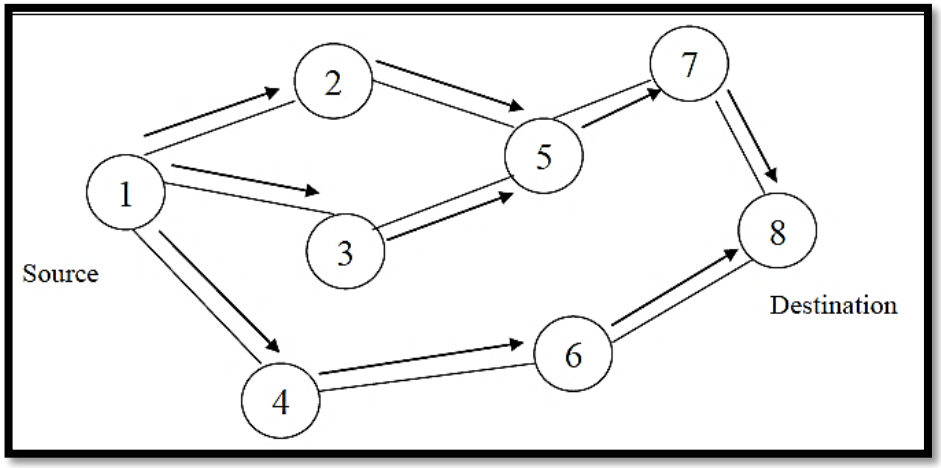

Figure 2.11: Propagation of route requests in AODV

Step 3: Destination node receiving the RREQ packet, it generates a RREP packet that consists of source node address, destination node address, destination sequence number, life time and hop count. The RREP is forwarded as unicast back to the source node using the reverse path. Figure 2.12 shows the RREP process by destination node 8.

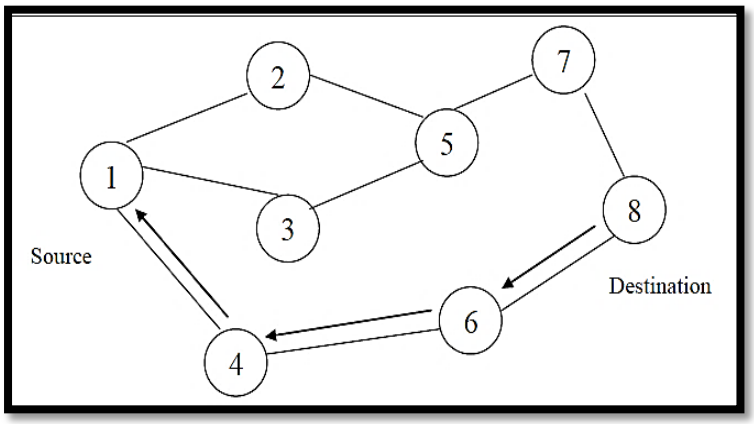

Figure 2.12: Propagation of route reply in AODV

Step 4: Source node begins transmission of data on receiving the first RREP packet and may start using another route if a new RREP packet arrives with a greater sequence number or has equal sequence number but a smaller hop count than the one previously in use.

Step 5: *Hello* packets are sent to neighbor nodes on an active transmission route to monitor link connectivity. If a link becomes broken, an RERR packet is sent.

AODV makes use of tables to store information for routing. The routing table stores information such as destination IP address, destination sequence number, next-hop address, life time (route expiration time), hop count, precursor list, network interface, valid destination sequence number flag, other state and routing flags [10]. Table 2.1 gives a possible routing table for node 4 from the example network topology illustrated in Figure 2.12.

Table 2.1: Routing table for node 4 in AODV

Routing table				
Destination	Sequence number	Next-hop	Hop count	Other fields
1	43	1	1	
2	26	1	2	
3	12	1	2	
4	65	4	0	
5	15	1	3	
6	23	6	1	
7	31	6	3	
8	27	6	2	

2.9 Black Hole Attack in AODV

MANETs are vulnerable to security attacks and AODV has no inbuilt security feature [11]. One of the common security attacks to routing protocols in MANETs is the black hole attack [12]. The operation of black hole in AODV is described in the following steps.

Step 1: Source node floods RREQ packet to all neighboring nodes when it desires a route to a destination node.

Step 2: Neighboring nodes receive the RREQ packet, check the routing tables for the next hop route to the destination node and if no such route is found, they also broadcast the RREQ packet to their own neighbors.

Step 3: When the malicious node receives this RREQ packet, it immediately sends a fake RREP packet to the source node and falsely sets its sequence number to a very high value.

Step 4: Source node compares the sequence numbers of the RREP packets received and uses the route from the RREP packet with the highest sequence number [13]. Since the RREP packet of malicious node comes with the highest sequence number, source node uses that route.

Step 5: Source node begins to forward data packet through the malicious node thinking the route is reliable.

Step 6: The malicious node drops the packets coming from the source and does not forward them to the destination node as required.

For example, in Figure 2.13, let us assume that node 2 becomes malicious which is represented as node M. When node 1 as the source node desires a route to the destination node, it broadcasts a RREP packet to its neighbors which are nodes 2 (M), 3 and 4. Node M without checking its routing table immediately sends a falsified RREP packet to node 1 with an abnormal high sequence number (99856745689) as shown in

the figure. Node 1 believes this is the fresher route since this has the highest sequence number and begins to forward data packet to M. On the other hand, nodes 3 and 4 rebroadcast the RREQ packet received from node 1 to nodes 5 and 6 respectively, since they do not have the next hop route entry to the destination node available in their routing table.

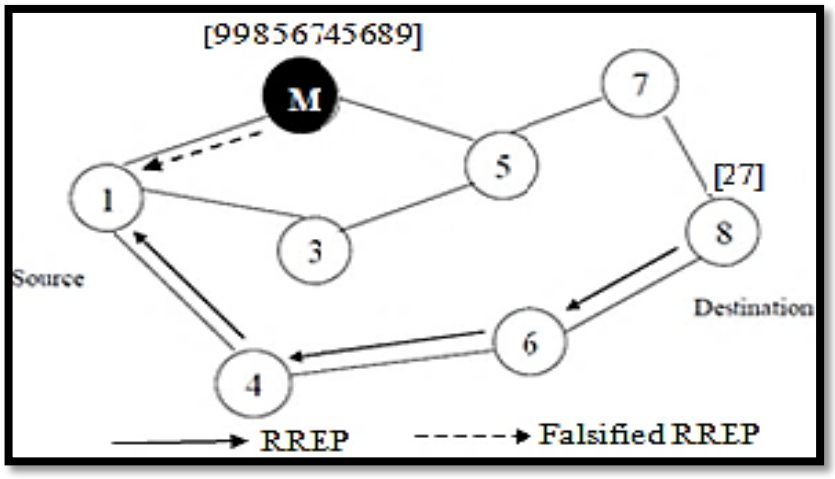

Figure 2.13: Black hole attack in AODV

After nodes 5, 6 and 7 rebroadcast the RREQ packet, node 8 as the destination node receives the RREQ packet and generates a RREP packet. It unicasts the RREP packet to node 1 with included sequence number as 27. When node 1 receives RREP packet from neighbor node 4, it compares the sequence number in RREP packet with the already existing malicious node sequence number in its routing table. It discards the RREP packet from node 4 and continues using the path through the malicious node M since the sequence number of RREP packet coming from malicious node M is higher (99856745689) than that of node 4 (27). Consequently, data packets forwarded to the malicious node are dropped.

2.10 Related Works

A lot of research has been made on black hole attack and several approaches have been proposed to tackle the issue of black hole attacks in MANETs. Some major works done on this topic are given below.

In [14], the authors proposed a scheme known as Association Based DSR protocol which encourages cooperation of nodes. It detects the selfish node and isolates it from participating in the routing process. This approach from the performance metric result obtained which was done varying the number of nodes shows a better throughput than normal DSR in the presence of black hole node but when no malicious node is present the normal DSR has less packet drop than this proposed scheme.

In [15], Tsou et al proposed an approach for DSR known as Baited-Black-hole DSR. In this scheme, the source node initially broadcasts a fake RREQ packet known as bait RREQ that contains a randomly selected and non-existent address as the desired destination. Since a black hole node does not check its routing table for a route to the destination, the baited malicious node immediately sends RREP packet to the source node claiming to have a route to the non-existent destination. Source node stores the detected black hole node id. After this process, source begins the actual route discovery process and is able to detect RREP packets from black hole node and discard such routes.

In [16], Pooja Jaiswal et al. proposed a solution to mitigate black hole attack in MANETs. In this approach, a mechanism was introduced to record destination sequence number of the neighbor nodes. The sequence numbers of the neighboring nodes are compared with that of the source node. If the difference between a source

node and a neighboring node is large, the neighboring node is considered to be malicious and its route entry is discarded. The result obtained shows a better metric performance for PDR and EED.

Lu et al. in [17] proposed a scheme to detect and avoid black hole in AODV called Secure AODV. It uses verification packets to ascertain the destination node as authentic via exchange of random numbers. When a source node receives a RREP packet, it sends a reply to the destination node using a verification secure RREQ packet that includes a randomly generated number by the source node. The destination node receives the secure RREQ packet and generates a secure RREP packet with the random number by the destination node. For the source node to determine a secure route, it waits for two or more secure RREP packets from different paths with the same random number. This scheme, however, fails to detect the malicious node when only a single secure RREP packet is received.

Raj et al. in [18] proposed an approach known as Detection, Prevention and Reactive AODV to identify and isolate malicious node in MANETs. In this approach, when a source node receives a RREP packet, the source node checks for the value of its sequence number on the routing table and also checks if the sequence number of the RREQ packet is higher than a defined threshold value. This threshold value is updated dynamically at a given time interval. If the RREP packet has a higher sequence number value than the threshold value, that node is considered malicious and blacklisted. An alarm packet is broadcasted to other nodes in the network so that RREP packet generated by the malicious node is discarded. This scheme is however affected by excessive overhead due to the update of the threshold value at regular time interval and the alarm control packet broadcasted.

In [19], the authors proposed an approach called Trust Based Dynamic Source Routing for black hole. Each node in the network observes its one hop distance neighbors during route discovery and records in a monitoring table their trust values which are based on rreq_fwd_credit. When a node receives a RREQ packet from a neighbor node, it stores the id of this neighbor node and increases the rreq_fwd_credit value for that node by 1 in its monitoring table therefore a node rreq_fwd_credit increases any time it forwards a RREQ packet and by this every node has gathers information about its surrounding neighbor and assigns a trust level which is either high or low which is based on the rreq_fwd_credit value for the node. A threshold value of 1 is selected for a rreq_fwd_credit value of a normal node. A black hole node never participates in RREQ packet forwarding. When an intermediate node receives a RREP packet, it attaches the trust value it recorded for the node it just receives the RREP packet and forwards it to next hop on the unicast path until it arrives at the source. The source node selects only the RREP packet with high trust values for all its nodes on the path.

Chapter 3

MODIFIED METHODOLOGY

3.1 Modified Scheme

The modified scheme utilizes promiscuous mode property of node in DSR for our improvement for both DSR and AODV protocols. Promiscuous mode enables a node to monitor packet traffic, intercept and read packets of neighbors in the same broadcast domain [20]. This implies that node can overhear its next neighbor and be able to tell if the received packet is forwarded along the route or not. For example in Figure 3.1, node B confirms that node C receives the data packet by overhearing C transmits the data packet to node D.

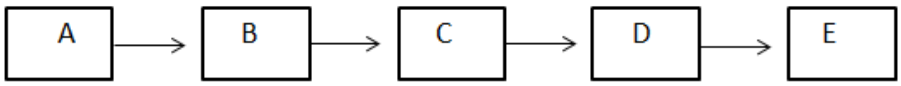

Figure 3.1: Hop to hop transfer of packet

In this scheme, the basic assumption is that there is no link and node failures and corrupted packets in the network. This implies that all nodes are functionig correctly apart from the node selected as the malicious node. All nodes in the network have an additional black hole list table created for storing malicious nodes detected in the network.

The proposed scheme increases network overhead due to continuous broadcast of alarm error packets to all nodes in the network for black hole list update when a node is

identified as a malicious node. Sometimes a false error packet can be generated that leads to declearing a non-malicious node as malicious in a situation when a node is temporarily unavailable.The following steps give a description of the implementation of this scheme.

Step 1: Initially routing cache and black hole list are empty for all nodes.

Step 2: Source node initiates route discovery so as to establish a route to the destination for communication as described in Section 2.4 and 2.8 for DSR and AODV respectively.

Step 3: After the route discovery is completed and the path to the destination is defined, source node checks if any of the intermediate or destination nodes is on the black hole list or not. If none of them is on the black hole list, data transmission begins, otherwise, a new route discovery process is started.

Step 4: During data transmission, nodes check if the next hop neighbor is the destination or not. If it is not then each node after forwarding the data packet starts overhearing in the promiscuous mode to track if the data packet is forwarded by the neighbor to the next intermediate node or not.

Step 5: If the node forwards the data packet to the next node then it means that the node is not malicious otherwise it is considered as a malicious node and error message is broadcasted to neighbors identifying the malicious node.

Step 6: All nodes update their black hole list by adding this malicious node to their black hole list.

Step 7: Source node removes the malicious route entry from its routing table and begins another route discovery process to get a new route for data transmission.

For the implementation of black hole attack and the modified scheme to mitigate the black hole attack, modification is made to the source code for DSR and AODV protocols. The two files where these modifications are made are dsragent.h and dsragent.cc for DSR protocol and aodv.h and aodv.cc for AODV protocol. These modifications are described and shown in subsequent sections.

3.2 Modification in DSR and AODV Protocols for Black Hole

For the addition of a black hole node in DSR and AODV protocols, a node needs to be declared as malicious. In dsragent.h and aodv.h files we modify the code as shown in the snippet of Figure 3.2 to declare a boolean variable malicious and black hole list which is presented in Appendix A.1 and A.2.

```
bool isMalicious;                  // this is the flag variable for the malicious node
bool IsMaliciousOrSelfish();       // this is the function for check the malicious node
bool CheckBlackList(int addr);     // this function for the black hole list
```

Figure 3.2: Codes to create bool variable for malicious node and black hole list

As explained in Chapter 2, during route discovery, the black hole increases the destination sequence number of its RREP packet in AODV to an abnormally high value and reduces the hop count to a small value so that the source node would forward the data packet to it as having a fresh route to the destination node. For the malicious node to act this way we modify the code as shown in Figure 3.3 which is presented in

Appendix B.2 by changing the sequence number to 99856745689 and hop count to 1 in aodv.cc file. During data transmission, the black hole node drops the data packet that it receives and does not forward them. As presented in Appendix B.2, Figure 3.4 shows the snippet of the C++ code added to both DSR and AODV protocol respectively for the malicious node to drop the packet.

```
sendReply(rq->rq_src,          // IP Destination
1,                             // Hop Count
rq->rq_dst,                    // Dest IP Address
99856745689,                   // Highest Dest Sequence Num
MY_ROUTE_TIMEOUT,              // Lifetime
rq->rq_timestamp);             // timestamp
```

Figure 3.3: Code to modify the sequence number and hop count in AODV

```
if(isMalicious==true)
{
drop(p,DROP_RTR_ROUTE_LOOP);
}
```

Figure 3.4: Code used by malicious node to drop packet in AODV and DSR

3.3 Modification of AODV and DSR for Modified Scheme

Since the modified scheme promiscuous mode property as explained in Section 3.1 exists in DSR only we create this promiscuous function in AODV as shown in Figure 3.5 which is presented in Appendix B.2. This function allows a node to have promiscuous mode functionality. This modification is made in aodv.cc file.

```
// to create the promiscuous function for aodv

else if (strcmp(argv[1], "install-tap") == 0) {
 mac_ = (Mac*)TclObject::lookup(argv[2]);
 if (mac_ == 0) return TCL_ERROR;
 mac_->installTap(this);
 return TCL_OK;
```

Figure 3.5: Snippet of the code to create the promiscuous mode function in AODV

According to the modified scheme, during data transmission, each node after forwarding the data packet to the next hop neighbor, it starts overhearing in the promiscuous mode to track if the data packet is forwarded by the neighbor or not. For each node to act in this manner, Figure 3.6 shows the code that is modified in AODV as presented in Appendix B.2.

```
void
AODV::tap(const Packet *p) {

 (index== node -> nodeid()_)

//listens to node about packet by overhearing method.

 /* snoop on the SR data */
  trace( net_id.dump(), cmh->uid());
     cmh->next_hop() = IP_BROADCAST;
       if(malicious_id == node_ -> nodeid()) {
          SendOutBCastErrorPkt(packet);
          ProcessDetectBlackhole;
           return true;
          }
  return false;
  }
```

Figure 3.6: Snippet of C++ code that allows nodes overhearing of neighbors in AODV

After the node discovers that its neighbor does not forward but drops the packet it receives, it alerts other nodes and the malicious node is added to the black hole list and route entry update is done. Figures 3.7 and 3.8 show the snippet of the code which is given in Appendix B.1 and B.2 respectively that allows nodes to update their black hole list and their route cache for DSR and routing table for AODV respectively

```
void DSRAgent::ProcessDetectBlackhole(SRPacket &p) {
        if (!isMalicious) {
                UpdateTheCache(p);
                m_blackList.push_back(p.src.addr);
                drop(p.pkt, DROP_RTR_TTL);
                return;
        }
}

bool DSRAgent::CheckBlackList(int addr) {
        for (std::list<int>::iterator it = m_blackList.begin(); it != m_blackList.end(); it ++) {
                if (*it == addr) return true;
        }

        return false;
```

Figure 3.7: Codes for adding the received information to the black hole list and update the routing cache in DSR

```
void
AODV::ProcessDetectBlackhole(SRPacket &p) {
        if (!isMalicious) {
                UpdateThert_table(p);
                m_blackList.push_back(p.src.addr);
                drop(p, DROP_RTR_ROUTE_LOOP);
                return;
        }
}

bool AODV::CheckBlackList(int addr) {
        for (std::list<int>::iterator it = m_blackList.begin(); it != m_blackList.end(); it ++) {
                if (*it == addr) return true;
        }
```

Figure 3.8: Codes for adding the received information to the black hole list and update the routing table in AODV

3.4 Performance Metrics

For this thesis, the performance metrics discussed below are used for the analysis and evaluation of the simulations.

3.4.1 Packet Delivery Ratio

This is a network performance metric that indicates the ratio of the number of packets received at the destination to the number of packets sent by the source as shown in (1). The greater the PDR value, the better the network performance is.

$$\text{PDR} = \frac{\text{Number of recieved packets}}{\text{Number of sent packets}} \quad (1)$$

3.4.2 Throughput

This performance metric indicates the rate of packets success fully transferred over a communication link at a given time in a network as shown in (2). It can be measured in bits per seconds (bps).

$$\text{Throughput} = \frac{\text{Number of packet sent} \times \text{Packet sent}}{\text{Simulation time}} \quad (2)$$

3.4.3 Average End-to-End Delay

The network metrics indicates the average time it would take the packets generated at the source to reach the destination as shown in (3). It is measured in seconds. EED delay for each received packet is computed by subtracting sending time of a packet from the receiving time of the packet and average EED is calculated for all packets as in (3).

$$\text{Average EED} = \frac{\sum(\text{Recieving time} - \text{Sending time})}{\text{Number of recieved packets}} \quad (3)$$

3.5 System Specifications

For the simulations, we used HP pavilion TS 15 notebook PC and installed oracle virtual box machine. The laptop has the following specifications: Intel(R) Core(TM) i5-4200U CPU @ 1.60 GHz (4 CPUs) 2.3 GHz processor, 12 GB RAM, 64-bit windows 10 operating system.

Chapter 4

SIMULATION RESULTS AND ANALYSIS

This chapter deals on the network simulator used called Network Simulator version-2 (NS-2), tools used in the network simulator, the simulation results and the analysis of the network performance obtain from the results.

4.1 NS-2 Network Simulator

NS-2 is a discrete event-driven, object-oriented network simulator [21]. It is a widely used open source software for network research which runs on Linux and was developed at the University of California, Berkeley. NS-2 is used for simulating and analyzing wireless and wired networks, and routing protocols. NS-2 is based on two programming languages namely C++ and Object Oriented Tool Command Language (OTCL) which is an extension of the Tool Command Language (TCL) and compatible with C++. Both languages are used for different reasons. C++ is used for implementing the protocol in detail and the OTCL is used for scheduling the events and controlling the simulation scenario for the user.

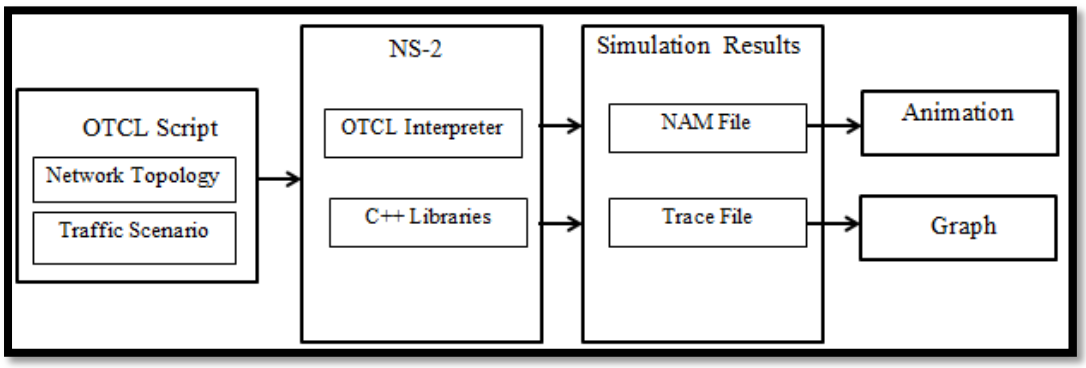

Figure 4.1: Simplified view of NS-2 process structure [21]

In the NS-2 process shown in Figure 4.1, the first stage is the declaration of the network topology, and traffic scenario. The configurations of these fields are created in the TCL script file which is programmed in C++. In the second stage, which is the simulation stage, the written OTCL script is interpreted by NS-2 in combination with C++ class libraries which are used for various common network protocols. For the third stage, NS-2 generates two analysis report files simultaneously after the TCL script of the simulation is interpreted. These two files are the Network Animator (NAM) file with the extension as .nam and trace file with the extension as .tr. The NAM shows the visual animated display of the nodes' behaviors during the simulation and the trace file shows the simulation traces of nodes in the format of texts.

4.2 AWK Script File

AWK is a powerful programming language interpretation tool made for processing text. It was developed in 1970 by Alfred Aho, Weinberger Peter, and Kernighan Brain [21]. Its main purpose is to extract data.

In this thesis, to collect the results from trace file which is in the format of text, an AWK script file is written which includes calculation of the network performance metrics such as PDR, throughput and average EED.

4.3 Simulation Model

For our simulations, NS-2.35 is installed [22] on oracle virtual box Windows-10 environment. As shown in Figure 4.2, the mobility and connection pattern is generated using setdest and cbrgen commands as presented in Appendix D.1. The mobility pattern describes the movement of nodes with their speeds and connection pattern deals with setting up traffic connection. Using the specified parameters mentioned in Chapter 5, we create our desired network using C++ codes in TCL script format as shown in Appendix C. The simulation is executed using the TCL script file to obtain a trace file

and nam file. To acquire the data analysis file for the network performance metrics of the network simulation which are PDR, throughput and average EED from the trace file generated, an AWK script called Performance.awk as presented in Appendix D.2 is used. The acquired network performance metric results are imported to Microsoft Excel 2010 for displaying in a graphical representation.

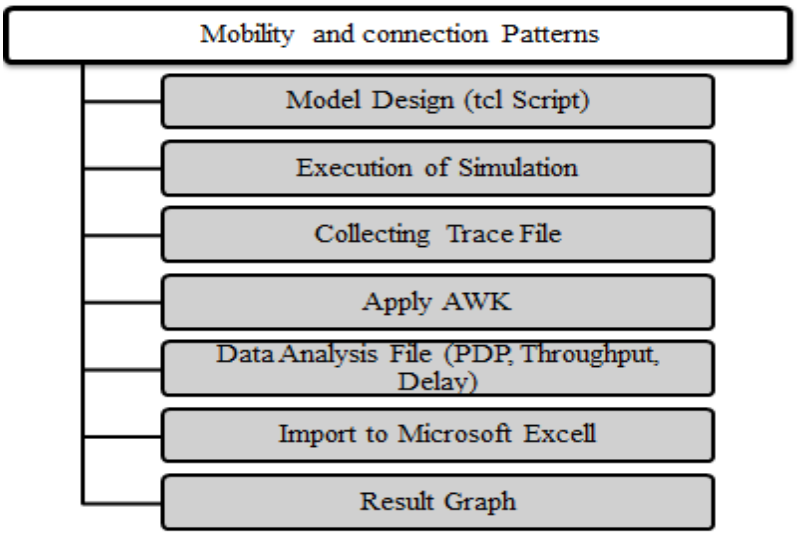

Figure 4.2: The simulation model [21]

4.4 Simulation Setup

The simulations are based on the analysis of DSR and AODV routing protocols with different simulation parameters as shown in Table 4.1. The implementation is to examine the effect of black hole attack on DSR and AODV based MANETs. The results in [13] are satisfyingly reproduced in this work. The thesis continues further with a modification for DSR and AODV protocols in the presence of black hole attack. The data analysis was conducted using the network performance metric discussed in Section 3.4 which are all measured against the various number of nodes. 20 simulation runs are done for each number of nodes and the average values are used to plot the graphs for each network performance metric.

Table 4.1: Simulation Parameters

Parameter	Settings
Network Routing Protocols	DSR, AODV
Traffic Type	CBR (UDP)
Mobility Model	Random Way Point
Network Area	670 × 670 m^2
Pause Time	0 seconds
Simulation Time	500 seconds
Packet Size	512 Bytes
Number of Runs	20
Maximum Speed	20 m/s
Numbers of Malicious Nodes	1
Number of Nodes	20, 40, 60, 80, 100

4.5 Simulation Scenarios

In this thesis, the DSR and AODV are the original protocols without black hole, Black hole DSR and Black hole AODV are protocols with the malicious node, and Modified DSR and Modified AODV are the modified protocols in the presence of the malicious node. All these protocols are implemented using the network parameters as shown in Table 4.1. Node 9 is made the malicious node which acts as the black hole node in the network. It sends back false RREP packet to the source and drops data packets whenever it receives them. Nodes in the network move randomly with the max speed as 20 m/s and the number of nodes is varied as 20, 40, 60, 80 and 100 with a fixed network area as 670 × 670 m^2.

4.5.1 Simulation Results for DSR Related Based MANETs

In this section, DSR protocol is investigated. Original DSR results are compared with Black hole DSR and Modified DSR. From the simulation results as shown in Figure 4.3 and Table 4.2, it can be observed that the PDR for DSR when there is no black hole ranges approximately from 0.991 to 0.999 for all nodes but when a malicious node is included as indicated for Black hole DSR, the PDR drops with values from 0.712 to 0.761 which indicates a decrease of around 26% which shows that data packets are

dropped by the black hole hence a decrease in the PDR of the network. From the result of the Modified DSR, it can be observed that modified protocol produces a better PDR even in the presence of the black hole with values ranging from 0.845 to 0.882 as indicating a 17% increase of the peak value than that of Black hole DSR which signifies less packet loss.

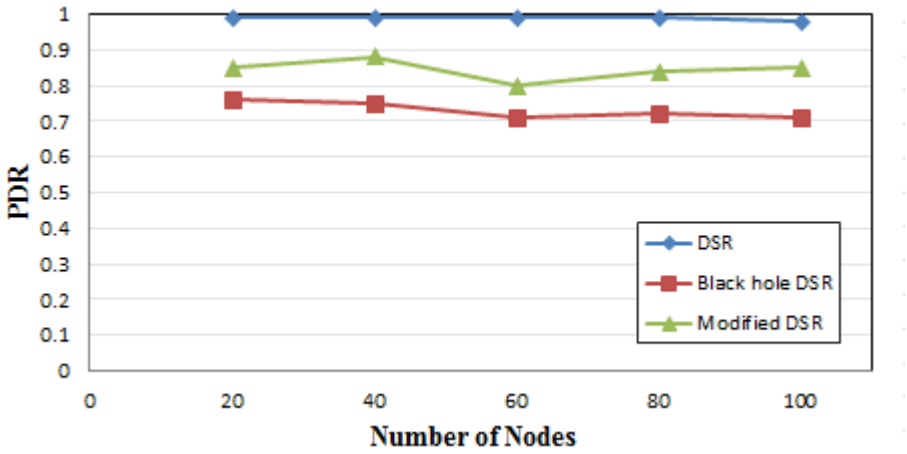

Figure 4.3: PDR with different number of nodes for DSR

Table 4.2: Average simulation results of PDR for DSR

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	0.991	0.999	0.999	0.999	0.989
Black hole DSR	0.761	0.751	0.715	0.721	0.712
Modified DSR	0.853	0.882	0.851	0.845	0.853

Figure 4.4 shows the throughput results for the simulation. It can be observed that Black hole DSR has a lower throughput in comparison to DSR and the Modified DSR. As seen in Table 4.3, DSR throughput values for all nodes are relatively high with values from 118.09 kbps to 118.79 kbps and the throughput for Black hole DSR decreases for all nodes with values ranging from 85.93 kbps to 92.80 kbps. This shows

a 25% decrease and is due to the fact that the average rate of data packet successfully sent over the communication channel in the network has been reduced by the black hole. There is a significant improvement in the Modified DSR with values from 99.62 kbps to 104.61 kbps for all nodes indicating approximately 14% increase of its peak value than that of the Black hole DSR.

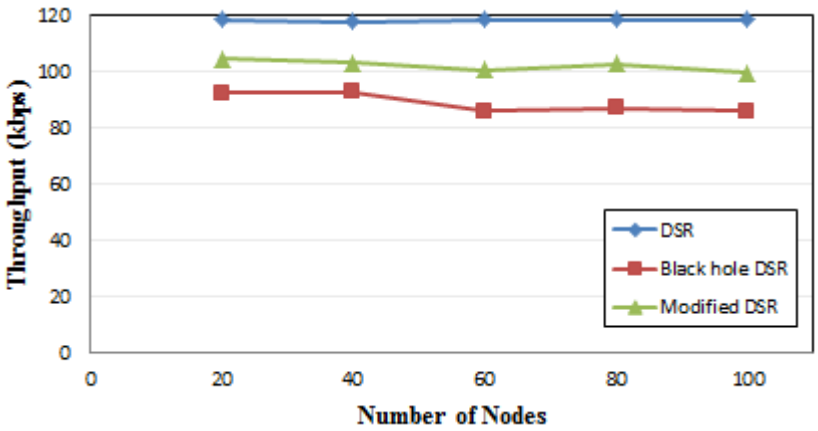

Figure 4.4: Throughput with different number of nodes for DSR

Table 4.3: Average simulation results of throughput in kbps for DSR

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	118.77	118.09	118.65	118.79	118.67
Black hole DSR	92.66	92.80	86.12	86.98	85.93
Modified DSR	104.61	103.09	100.71	102.86	99.62

Figure 4.5 shows the results of the average EED of the network which is measured in milliseconds (ms) when varying the number of nodes and it can be observed that the average EED for the Black hole DSR is lower in comparison to the others. As seen in Table 4.4, the average EED for DSR for all the nodes ranges from 162.23 ms to 193.24 ms and in the presence of the black hole attack decreases to between 100.52 ms to 135.78 ms which indicates approximately a 32% decrease. This decrease is as a result

of the shortening of the route discovery process by the black hole since the source begins to transmit data as soon as it receives the falsified RREP packet from the malicious node. Furthermore, since there is less traffic in the network because of dropped packets, the few packets which are able to avoid the malicious node gets to the destination faster thereby reducing the average EED of the packets received. The Modified DSR also has a lower average EED than the original DSR without black hole with values from 150.66 to 173.43 indicating around 10% decrease. It should be noted that the decrease of average EED for Black hole DSR is not an improvent in the system performance but it is as a result of high rate of dropped packets by the black hole.

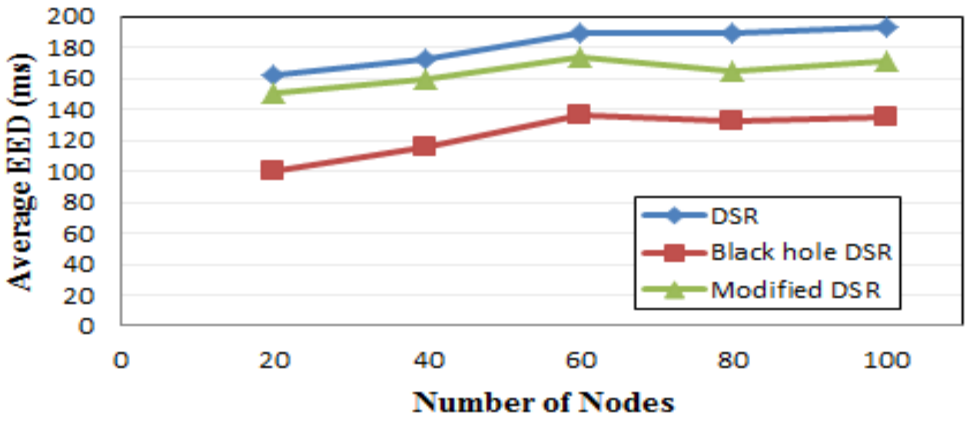

Figure 4.5: Average EED with different number of nodes for DSR

Table 4.4: Average simulation results average EED in ms for DSR

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	169.23	173.26	189.51	188.78	193.24
Black hole DSR	100.52	116.32	135.78	133.42	134.55
Modified DSR	150.66	159.68	173.43	165.42	170.91

4.5.2 Simulation Results for AODV Related Based MANETs

In this section, AODV protocol is investigated. Original AODV results are compared with Black hole and Modified AODV. It can be seen from Figure 4.6 that the PDR and throughput values for the various number of nodes for each protocol are just slightly different and the black hole greatly decreases the PDR and throughput for AODV based MANETs. As it can be observed in Table 4.5, the PDR for AODV without black hole is relatively high with peak value as 0.999 but when a malicious node is present as implemented for Black hole AODV, the PDR is brought down for all the various number nodes ranging between 0.162 to 0.187. This shows that the PDR decreases by 82% since black hole node forces a packet to be sent to it by including an abnormal high sequence number in its RREP packet. The Modified AODV results show better PDR values ranging from 0.814 to 0.853 which shows around 79% increase than Black hole AODV. This indicates less packet loss in comparison to Black hole AODV since the Modified AODV can detect the node that is malicious and then use another secured route for data transmission

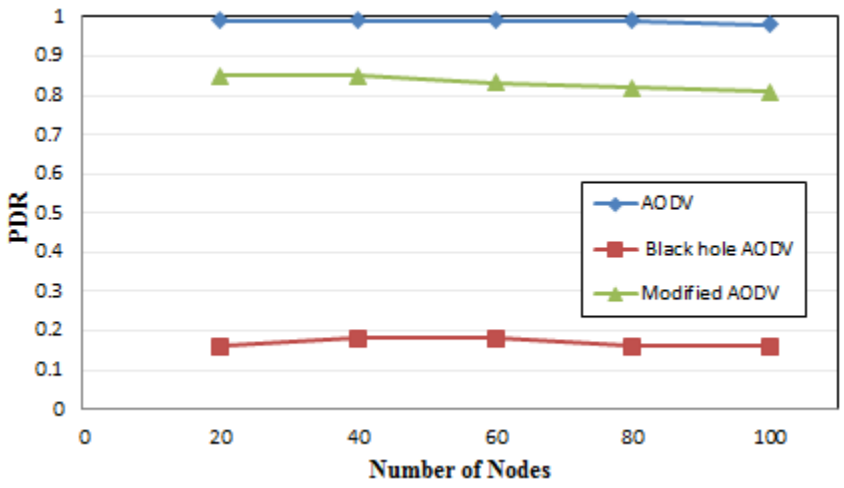

Figure 4.6: PDR with different number of nodes for AODV

Table 4.5: Average simulation results of PDR for AODV

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	0.996	0.999	0.998	0.996	0.984
Black hole DSR	0.162	0.182	0.187	0.162	0.161
Modified DSR	0.852	0.853	0.832	0.825	0.814

Figure 4.7 indicates the resulting graph of the network throughput for the simulation while varying the number of nodes. It can be observed that Black hole AODV shows a lower throughput for the different network load when compared with the AODV without black hole and the Modified AODV. As shown in Table 4.6, AODV throughput values for the different number of nodes range from 115.65 kbps to 117.9 kbps with the highest value as 117.9 kbps and the throughput for Black hole AODV decreases with values from 15.69 kbps to 18.21 kbps indicating around 85% reduction in comparison to AODV without black hole. This is due to the fact that the average rate of successfully data packet sent over the network has been reduced and most packet are discarded by the black hole node. The Modified AODV outperforms the AODV with black hole with values ranging from 94.84 kbps to 98.25 indicating around 81% increase.

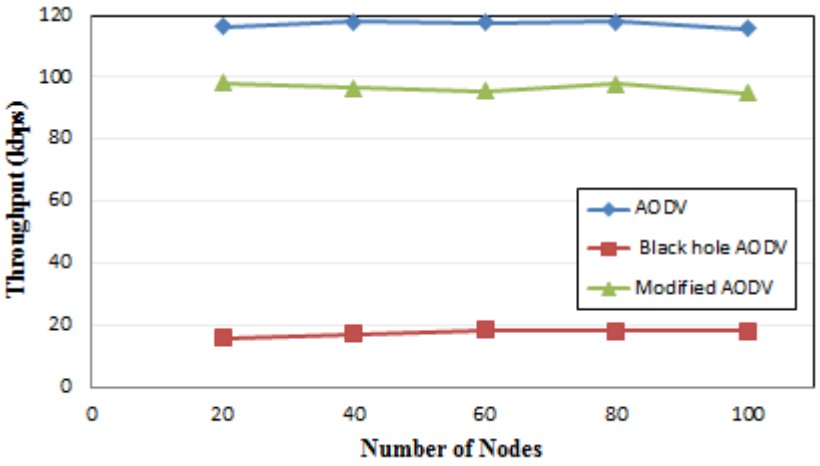

Figure 4.7: Throughput with different number of nodes for AODV

Table 4.6: Average simulation results of throughput in kbps for AODV

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	116.40	117.32	117.56	117.90	115.65
Black hole DSR	15.69	17.20	18.21	17.92	17.95
Modified DSR	98.25	96.53	95.54	97.83	94.84

From Figure 5.6, it is observed that the results of the average EED of the network while varying the network load significantly decreases for the Black hole AODV in comparison to the others. Table 5.7 shows that the average EED for Black hole AODV decreasing with approximately 92% in comparison to AODV without black hole. For AODV without black hole, the average EED for all nodes ranges from 108.89 ms to 119.51 ms and the Black hole AODV ranges from 17.28 ms to 20.84 ms. The Modified AODV shows a lower average EED than the AODV without black hole with around 17%.

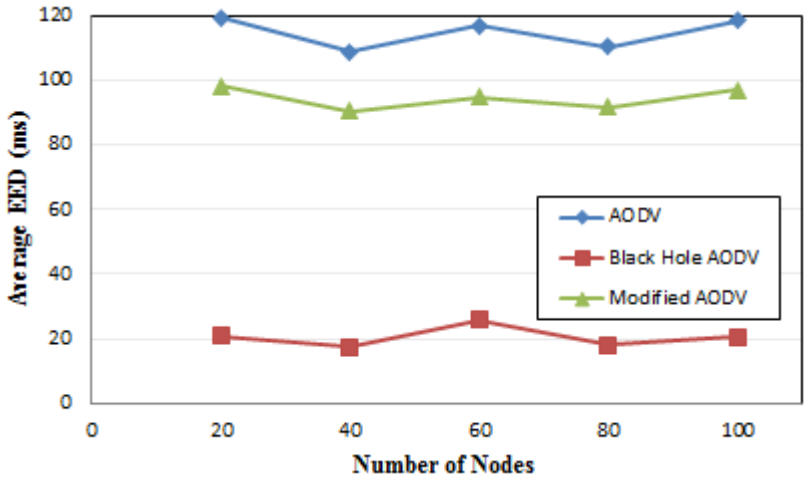

Figure 4.8: Average EED with different number of nodes for AODV

Table 4.7: Average simulation results of average EED in ms for AODV

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	119.51	108.89	116.94	110.36	118.51
Black hole DSR	20.84	17.28	25.92	17.92	20.59
Modified DSR	98.47	90.55	94.83	91.68	97.18

5.2.3 Comparison of Simulation Results for DSR and AODV

The performance of DSR and AODV are shown by the results obtained from the simulations while varying the numbers of nodes. The results show the impacts of the black hole and the modification on both protocols using PDR, throughput and average EED. The comparison for each of these protocol on the different metrics are analysed below based on the tables given.

Table 4.8: PDR for DSR and AODV

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	0.991	0.999	0.999	0.999	0.999
AODV	0.996	0.999	0.998	0.996	0.984
Black hole DSR	0.761	0.751	0.715	0.721	0.712
Black hole AODV	0.162	0.182	0.187	0.162	0.161
Modified DSR	0.853	0.882	0.852	0.845	0.853
Modified AODV	0.852	0.853	0.832	0.825	0.814

We observe from the results presented in Table 4.8 that the PDR for both original DSR and AODV is relatively high and remains almost the same despite the different number of nodes with values ranging from 0.991 to 0.999 for both but when a malicious node is introduced, PDR performance of DSR decreases approximately by 26% as seen in Black hole DSR and PDR for AODV decreases by around 82% as seen also in Black hole AODV indicating that AODV is more affected by the black hole since the malicious node makes use of hop count and the sequence number option in AODV to force the selection of its route in the RREP packet. The modified approach shows improvement for both protocols with Modified DSR increasing to values ranging from 0.845 to 0.882 and Modified AODV increasing with values ranging from 0.814 to 0.853 which is due to the fact that malicious node is being identified and avoided during route selection. Similarly in [13], PDR of DSR for all number of nodes falls approximately between 0.96 and 0.99 and when black hole is introduced, it drops to around 0.68 to 0.75. For AODV, PDR ranges approximately from 0.95 to 0.99 when there is no black hole and drops significantly when black hole is included to around 0.08 to 0.12 which is similar to the results obtained in this thesis study.

Table 4.9: Throughput in kbps for DSR and AODV

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	118.77	118.09	118.65	118.79	118.67
AODV	116.40	117.32	117.56	117.90	115.65
Black hole DSR	92.66	92.80	86.12	86.98	85.93
Black hole AODV	15.69	17.20	18.21	17.92	17.95
Modified DSR	104.61	103.09	100.71	102.86	99.62
Modified AODV	98.25	96.53	95.54	97.83	94.84

It can be seen from Table 5.9 that the throughput remains almost constant for the different number of nodes for both DSR and AODV protocols. The results also indicates that when black hole attack is launched, throughput values for DSR decrease with values ranging from 85.93 kbps to 92.80 kbps showing around 25% drop in comparison to DSR without black hole having values from 118.09 kbps to 118.79 kbps and AODV with black hole decreases with values from 15.69 kbps to 18.21 kbps showing around 85% drop as against AODV without black hole. This indicates also that black hole attack has greater impact on AODV. The modified DSR and AODV improves the network throughput even in the presence of black hole attack as Modified DSR increases with values ranging from 99.62 kbps to 104.61 kbps and Modified AODV increase with values ranging from 94.84 kbps to 98.25 kbps. Similarly also in [13], the throughput of DSR falls approximately between 132 kbps and 138 kbps for all nodes and when black hole is introduced, it decreases to around 90 kbps to 97 kbps. For AODV when there is no black hole, the values ranges approximately from 130 kbps to 135 kbps and drops significantly when black hole is included to around 6 kbps to 9 kbps which shows a similar drop as the simulation results obtained in this thesis study.

Table 4.10: Average EED in ms for DSR and AODV

Protocol	Number of Nodes				
	20	40	60	80	100
DSR	162.23	173.22	189.51	188.78	193.24
AODV	119.51	108.89	116.94	110.36	118.51
Black hole DSR	100.52	116.32	135.78	133.14	134.55
Black hole AODV	20.84	17.28	25.92	17.92	20.59
Modified DSR	150.66	159.68	173.42	165.42	170.91
Modified AODV	98.47	90.55	94.83	91.68	97.18

It is observed from the results showing in Table 4.10 that DSR has a higher average EED than AODV which is due to multiple route cache overhead found in DSR. When black hole is included to DSR and AODV protocols, the average EED of DSR decreases for all nodes with values ranging from 100.52 ms to 135.78 ms as against DSR without black hole having values ranging from 162.23 ms to 193.24 ms having around 32% decrease while AODV decreases significantly with values for the different nodes ranging from 17.28 ms to 25.92 ms as against AODV without black hole having its values ranging from 108.89 ms to 119.51 ms signifying a decrease of around 82% which is due to the fact that AODV is highly impacted by the malicious node and hence the data traffic is greatly reduce making successfully transferred data packets to take shorter time to get to their respective destination. The modified DSR and AODV having been able to mitigate the effect of the black hole attack shows a lower average EED with around 10% and 17% decrease respectively than original DSR and AODV.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

MANETs are easily prone to security attacks with black hole attack as one of the most common security attack. In this thesis, our main aim is to investigate the effect of the black hole attack and discuss a method to lessen the effect of the black hole attack on MANETs. Firstly, mechanism of DSR and AODV protocols is tackled. Secondly, the behavior of DSR and AODV protocols in the presence of black hole attack is studied with a review on previous work done on black hole prevention. Thirdly, a modified approach for DSR and AODV protocols to improve the network performances even in the presence of a black hole attack is discussed. Finally implementations for original DSR and AODV, Black hole DSR and AODV, and Modified DSR and AODV protocols are carried out so examine the effect of the black hole on these protocols and to see the improvement of modified protocols in the network.

From the results of the simulations done, it can be observed that PDR, throughput, and average EED decrease for both DSR and AODV protocols when a black hole is included. It can be observed that the PDR and throughput remain almost the same irrespective of the network load. Results show that for DSR based network, when black hole is introduced, the values for PDR decrease by 26%, throughput decrease by 25%, and average EED decrease by 32%. For AODV based network, when a black hole is introduced, the PDR decreases averagely by 82%, throughput decreases by 85%, and

average EED decreases by 82%. Finally, the modified protocols show an improvement to original DSR and AODV in the presence of black attack since they are able to detect the black hole node in the network by overhearing in promiscuous mode to neighbors if the received packet is forwarded or not. Consequently, the malicious node is stored in a black hole list and subsequently source avoids the use of any path having the malicious node. The simulation results of protocols show that even with black hole in the Modified DSR, the PDR increases with 17%, throughput with 14% higher than Black hole DSR and 10% lower for average EED than DSR without black hole. For Modified AODV, PDR increases with 79%, throughput with 81% higher than Black hole AODV and 17% lower for average EED than AODV without black hole.

In conclusion, it can be said of the results generated from the simulations that black hole attack decreases the overall network performance and it has a more severe impact on AODV protocol than on DSR protocol and the modified protocols outperforms original DSR and AODV protocols when black hole attack is launched thereby improving the security and enhancing the network performance of DSR and AODV protocols against black hole attacks.

5.2 Future Work

For the future work in relation to this thesis, we intend to fix the fault tolerance such that a node not forwarding packet due to its temporal unavailability is not falsefully considered as a malicious node. We will extend our work by implementing and analyzing our modified approach on proactive and hybrid routing protocols due to their on-demand nature and also investigate a situation of two or more black holes in the network.

REFERENCES

[1] Ghosekar, P., Katkar, G., & Ghorpade, P. (2010). Mobile Ad Hoc Networking: Imperatives and Challenges. *IJCA Special Issue on Mobile Ad-hoc Networks (MANETs).*

[2] Aarti, D. S. (2013). Study of Manet: Characteristics, Challenges, Application and Security Attacks. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(5), pp. 252-257.

[3] Dhenakaran S., & Parvathavarthini A. (2013) An Overview of Routing Protocols in Mobile Ad-Hoc Network, *International Journal of Research in Computer Science and Software Engineering*, 3(2).

[4] Bilandi, N., & Verma, H. K. (2012). Comparative Analysis of Reactive, Proactive and Hybrid Routing Protocols in MANET. *International Journal of Electronics and Computer Science Engineering*, 1(3), pp. 1660-1667.

[5] Johnson, D., Hu, Y., & Maltz, D. (2007). The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4. *IETF Internet draft*, No. RFC 4728.

[6] Johnson D. B., Maltz D. A., & Broch J. (2001). DSR: The Dynamic Source Routing Protocol for Multi-hop Wireless Ad Hoc Networks". *In C. E. Perkins, editor, Ad Hoc Networking*, pp. 139–172.

[7] Abdelshafy, M. A., & King, P. J. (2015). Dynamic Source Routing under Attacks. *In Reliable Networks Design and Modeling (RNDM), 7th International Workshop*, pp. 174-180.

[8] Bhardwaj, A. (2014). Secure Routing in DSR to Mitigate Black Hole Attack. In *Control, Instrumentation, Communication and Computational Technologies (ICCICCT) International Conference*, pp. 985-989.

[9] Woungang, I., Dhurandher, S. K., Peddi, R. D., & Obaidat, M. S. (2012). Detecting Blackhole Attacks on DSR-based Mobile Ad Hoc Networks. In *Computer, Information and Telecommunication Systems (CITS), 2012 International Conference,* pp. 1-5.

[10] Perkins, C., Belding-Royer, E., & Das, S. (2003). Ad Hoc On-demand Distance Vector (AODV) Routing. *IETF MANET Working Group AODV Draft*, No. RFC 3561.

[11] Jali, K. A., Ahmad, Z., & Ab Manan, J. L. (2011). Mitigation of Black Hole Attacks for AODV Routing Protocol. *International Journal of New Computer Architectures and their Applications*, 1(2), pp. 336-343.

[12] Tseng, F. H., Chou, L. D., & Chao, H. C. (2011). A Survey of Black Hole Attacks in Wireless Mobile Ad Hoc Networks. *Human-centric Computing and Information Sciences,* 1(1), pp. 1-16.

[13] Mejaele L. & Ochola E. O. (2014). AODV vs. DSR: Simulation-Based Comparison of Ad-hoc Network Reactive Protocols under Black Hole Attack.

in *Proc. of the Second Intl. Conference on Advances in Computing, Electronics and Electrical Technology*.

[14] Bhalaji, N., & Shanmugam, A. (2009). Association between Nodes to Combat Blackhole Attack in DSR based MANET. In *Wireless and Optical Communications Networks (WOCN'09) IFIP International Conference*, pp. 1-5.

[15] Tsou, P. C., Chang, J. M., Lin, Y. H., Chao, H. C., & Chen, J. L. (2011). Developing a BDSR Scheme to Avoid Black Hole Attack based on Proactive and Reactive Architecture in MANETs. In *Advanced Communication Technology (ICACT) 13th International Conference*, pp. 755-760.

[16] Jaiswal, P., & Kumar, D. R. (2012). Prevention of Black Hole Attack in MANET. *International Journal of Computer Networks and Wireless Communications*, *2*(5).

[17] Lu, S., Li, L., Lam, K. Y., & Jia, L. (2009). SAODV: A MANET Routing Protocol that can Withstand Black Hole Attack. In *International Conference on Computational Intelligence and Security*, pp. 421-425.

[18] Raj, P. N., & Swadas, P. B. (2009). DPRAODV: A Dyanamic Learning System against Blackhole Attack in AODV based MANET. *arXiv preprint arXiv:0909.2371*.

[19] Mohanapriya, M., & Krishnamurthi, I. (2014). Trust based DSR Routing Protocol for Mitigating Cooperative Black Hole Attacks in Ad Hoc Networks. *Arabian Journal for Science and Engineering*, *39*(3), pp. 1825-1833.

[20] Singh P. K., & Sharma G. (2012). An Efficient Prevention of Black Hole Problem in AODV Routing Protocol in MANET, *11th International Conference on Trust, Security and Privacy in Computing and Communications*.

[21] Issariyatul T., Hossain E. (2013), Introduction to Network Simulator NS-2, *Springer, 2nd Edition*.

[22] "How to Install NS-2.35 in Ubuntu-13.10/14.04", available at http://installwithme.blogspot.com/2015/05/how-to-install-ns-2.35-in-ubuntu-13.10-or-14.04.html (last accessed on November 2015).

APPENDICES

Appendix A: Script Files (.h)

Appendix A.1: DSR Script (dsragent.h) Original DSR Script file is modified (modified parts are provided in boxes).

```
  */
#ifndef _DSRAgent_h
#define _DSRAgent_h
class DSRAgent;
#include <stdarg.h>

#include <object.h>
#include <agent.h>
#include <trace.h>
#include <packet.h>
#include <dsr-priqueue.h>
#include <mac.h>
#include <mobilenode.h>
#include <list>

#include "path.h"
#include "srpacket.h"
#include "routecache.h"
#include "requesttable.h"
#include "flowstruct.h"

#define BUFFER_CHECK 0.03       // seconds between buffer checks
#define RREQ_JITTER 0.010       // seconds to jitter broadcast route requests
#define SEND_TIMEOUT 30.0       // # seconds a packet can live in sendbuf
#define SEND_BUF_SIZE 64
#define RTREP_HOLDOFF_SIZE 10
…
class SendBufferTimer : public TimerHandler {
public:
        SendBufferTimer(DSRAgent *a) : TimerHandler() { a_ = a; }
        void expire(Event *e);
protected:
        DSRAgent *a_;
};

LIST_HEAD(DSRAgent_List, DSRAgent);

class DSRAgent : public Tap, public Agent {
public:
        virtual int command(int argc, const char*const* argv);
        virtual void recv(Packet*, Handler* callback = 0);

        void tap(const Packet *p);
        // tap out all data packets received at this host and promiscously snoop
        // them for interesting tidbits
```

```
        void Terminate(void);
        // called at the end of the simulation to purge all packets
        void sendOutBCastPkt(Packet *p);

        DSRAgent();
        ~DSRAgent();

private:
        Trace *logtarget;
        int off_mac_;
        int off_ll_;
        int off_ip_;
        int off_sr_;

bool isMalicious;                       // this is the flag variable for the malicious node
bool IsMaliciousOrSelfish();        // this is the function for check the malicious node
bool CheckBlackList(int addr);

        // will eventually need to handle multiple infs, but this is okay for
        // now 1/28/98 -dam
        ID net_id, MAC_id;              // our IP addr and MAC addr
        NsObject *ll;                   // our link layer output
        CMUPriQueue *ifq;               // output interface queue
        Mac *mac_;

        // extensions for wired cum wireless sim mode
        MobileNode *node_;
        int diff_subnet(ID dest, ID myid);

        // extensions for mobileIP
        NsObject *port_dmux_;    // my port dmux
        std::list<int> m_blackList;
         *  A cache of recently seen packets on the TAP so that I
         *  don't process them over and over again.
         */
        int tap_uid_cache[TAP_CACHE_SIZE];
#endif

#endif // _DSRAgent_h
```

Appendix A.2: AODV Script (aodv.h) Original AODV Script file is modified (modified parts are provided in boxes).

```
*/
#ifndef __aodv_h__
#define __aodv_h__

//#include <agent.h>
//#include <packet.h>
//#include <sys/types.h>
//#include <cmu/list.h>
//#include <scheduler.h>

#include <cmu-trace.h>
#include <priqueue.h>
#include <aodv/aodv_rtable.h>
#include <aodv/aodv_rqueue.h>
#include <classifier/classifier-port.h>
#include <mac.h>
/*
  Allows local repair of routes
*/
#define AODV_LOCAL_REPAIR

  Allows AODV to use link-layer (802.11) feedback in determining when
  links are up/down.
*/
#define AODV_LINK_LAYER_DETECTION
/*
  Causes AODV to apply a "smoothing" function to the link layer feedback
  that is generated by 802.11.  In essence, it requires that RT_MAX_ERROR
  errors occurs within a window of RT_MAX_ERROR_TIME before the link
  is considered bad.
*/
  Timers (Broadcast ID, Hello, Neighbor Cache, Route Cache)
*/
class BroadcastTimer : public Handler {
public:
      BroadcastTimer(AODV* a) : agent(a) {}
      void        handle(Event*);
private:
      AODV    *agent;
         Event   intr;
};

class HelloTimer : public Handler {
public:
      HelloTimer(AODV* a) : agent(a) {}
      void        handle(Event*);
private:
```

```
        AODV    *agent;
          Event   intr;
};

class NeighborTimer : public Handler {
public:
        NeighborTimer(AODV* a) : agent(a) {}
        void        handle(Event*);
private:
        AODV    *agent;
          Event   intr;
};

class RouteCacheTimer : public Handler {
public:
        RouteCacheTimer(AODV* a) : agent(a) {}
        void        handle(Event*);
private:
        AODV    *agent;
          Event   intr;
};

*/
class BroadcastID {
        friend class AODV;
 public:
        BroadcastID(nsaddr_t i, u_int32_t b) { src = i; id = b;  }
 protected:
        LIST_ENTRY(BroadcastID) link;
        nsaddr_t        src;
        u_int32_t       id;
        double          expire;         // now + BCAST_ID_SAVE s
};
LIST_HEAD(aodv_bcache, BroadcastID);
/*
  The Routing Agent
*/
class AODV: public Tap, public Agent {
   * make some friends first
   */
        friend class aodv_rt_entry;
        friend class BroadcastTimer;
        friend class HelloTimer;
        friend class NeighborTimer;
        friend class RouteCacheTimer;
        friend class LocalRepairTimer;

 public:
        void tap(const Packet *p);
        AODV(nsaddr_t id);
        void                    recv(Packet *p, Handler *);
```

```
protected:
        Mac *mac_;
        int             command(int, const char *const *);
        int             initialized() { return 1 && target_; }
        /*
         * Route Table Management
         */
        void            rt_resolve(Packet *p);
        void            rt_update(aodv_rt_entry *rt, u_int32_t seqnum,
                                  u_int16_t metric, nsaddr_t nexthop,
                                  double expire_time);
        void            rt_down(aodv_rt_entry *rt);
        void            local_rt_repair(aodv_rt_entry *rt, Packet *p);
public:
        void            rt_ll_failed(Packet *p);
        void            handle_link_failure(nsaddr_t id);
protected:
        void            rt_purge(void);

        void            enque(aodv_rt_entry *rt, Packet *p);
        Packet*         deque(aodv_rt_entry *rt);

        bool isMalicious;                   // this is the flag variable for the malicious node
        bool IsMaliciousOrSelfish();        // this is the function for check the malicious node
        bool CheckBlackList(int addr);

         * Neighbor Management
         */
        void            nb_insert(nsaddr_t id);
        AODV_Neighbor*  nb_lookup(nsaddr_t id);
        void            nb_delete(nsaddr_t id);
        void            nb_purge(void);

#endif /* __aodv_h__ */
```

Appendix B: Script Files (.cc)

Appendix B.1: DSR Script (dsragent.cc) Original DSR Script file is modified (modified parts are provided in boxes).

```
extern "C" {
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <signal.h>
#include <float.h>
}
#include <object.h>
#include <agent.h>
#include <trace.h>
#include <packet.h>
#include <scheduler.h>
#include <random.h>

#include <mac.h>
#include <ll.h>
#include <cmu-trace.h>

#include "path.h"
#include "srpacket.h"
#include "routecache.h"
#include "requesttable.h"
#include "dsragent.h"
….
static class DSRAgentClass : public TclClass {
public:
        DSRAgentClass() : TclClass("Agent/DSRAgent") {}
        TclObject* create(int, const char*const*) {
                return (new DSRAgent);
        }
} class_DSRAgent;

int
DSRAgent::command(int argc, const char*const* argv)
{
        TclObject *obj;
        if (argc == 2)
        {
                if (strcasecmp(argv[1], "testinit") == 0)
                {
                        testinit();
                        return TCL_OK;
                }
                if (strcasecmp(argv[1], "reset") == 0)
                {
```

```
        Terminate();
        return Agent::command(argc, argv);
}
if (strcasecmp(argv[1], "check-cache") == 0)
{
        return route_cache->command(argc, argv);
}
if (strcasecmp(argv[1], "startdsr") == 0)
{
        // cheap source of jitter
        send_buf_timer.sched(BUFFER_CHECK
                + BUFFER_CHECK * Random::uniform(1.0));
        return route_cache->command(argc, argv);
}
else if (strcasecmp(argv[1], "send_timeout") == 0)
{
        send_timeout = strtod(argv[2], NULL);
        return TCL_OK;
}

if ((obj = TclObject::lookup(argv[2])) == 0)
{
        fprintf(stderr, "DSRAgent: %s lookup of %s failed\n", argv[1],
                argv[2]);
        return TCL_ERROR;
}

if (strcasecmp(argv[1], "log-target") == 0)  {
        logtarget = (Trace*)obj;
        return route_cache->command(argc, argv);
}
else if (strcasecmp(argv[1], "tracetarget") == 0)
{
        logtarget = (Trace*)obj;
        return route_cache->command(argc, argv);
}
else if (strcasecmp(argv[1], "install-tap") == 0)
{
        mac_ = (Mac*)obj;
        mac_->installTap(this);
        return TCL_OK;
}
else if (strcasecmp(argv[1], "node") == 0)
{
        node_ = (MobileNode *)obj;
        int node_id = node_->nodeid();
        if(IsMaliciousOrSelfish()) isMalicious = true;
        else isMalicious = false;

        return TCL_OK;
}
```

```
		else if (strcasecmp(argv[1], "port-dmux") == 0)
		{
			port_dmux_ = (NsObject *)obj;
			return TCL_OK;
		}
	}
	else if (argc == 4)
	{
		if (strcasecmp(argv[1], "add-ll") == 0)
		{
			if ((obj = TclObject::lookup(argv[2])) == 0) {
				fprintf(stderr, "DSRAgent: %s lookup of %s failed\n",
argv[1],
					argv[2]);
				return TCL_ERROR;
			}
			ll = (NsObject*)obj;
			if ((obj = TclObject::lookup(argv[3])) == 0) {
				fprintf(stderr, "DSRAgent: %s lookup of %s failed\n",
argv[1],
					argv[3]);
				return TCL_ERROR;
			}
			ifq = (CMUPriQueue *)obj;
			return TCL_OK;
		}
	}
	return Agent::command(argc, argv);
}

bool DSRAgent::IsMaliciousOrSelfish() {
	FILE* fp = NULL;

	// please input the real path of node information file
	fp = fopen("/home/ailem/node.info", "r+");
	if(fp == NULL) {
		printf("Please input the node information file path correctly!\n");
		return false;
	}
	int malicious_id = -1;

	while(!feof(fp)) {
		fscanf(fp, "%d\n", &malicious_id);

		if(malicious_id == node_ -> nodeid()) {
			printf("NodeId: %d is Malicious!\n", malicious_id);
			return true;
		}
	}
	return false;
}
```

```
void DSRAgent::SendOutBCastErrorPkt(Packet *p)
{
        hdr_cmn *cmh = hdr_cmn::access(p);
        cmh->ptype() = PT_DSR_ERROR;
        if (cmh->direction() == hdr_cmn::UP) cmh->direction() = hdr_cmn::DOWN;
        Scheduler::instance().schedule(ll, p, 0.0);
        return;
}
/* This is the function for broadcast the error packet to the others. */
void DSRAgent::sendOutBCastPkt(Packet *p)
{
        hdr_cmn *cmh = hdr_cmn::access(p);
        if (cmh->direction() == hdr_cmn::UP) cmh->direction() = hdr_cmn::DOWN;
        Scheduler::instance().schedule(ll, p, 0.0);
        return;
}
void DSRAgent::ProcessDetectBlackhole(SRPacket &p) {
        if (!isMalicious) {
                UpdateTheCache(p);
                m_blackList.push_back(p.src.addr);
                drop(p.pkt, DROP_RTR_TTL);
                return;
        }
}
bool DSRAgent::CheckBlackList(int addr) {
        for (std::list<int>::iterator it = m_blackList.begin(); it != m_blackList.end(); it
++) {
                if (*it == addr) return true;
        }
        return false;
}
/* This is the Update the cache after receive the error notification. */
void DSRAgent::UpdateTheCache(SRPacket& p)
{
        hdr_sr *srh = hdr_sr::access(p.pkt);
        assert(srh->num_route_errors() > 0);
        for (int c = 0; c < srh->num_route_errors(); c++)
        {
                assert(srh->down_links()[c].addr_type == NS_AF_INET);

                route_cache->updateBrokenLink(ID(srh->down_links()[c].from_addr,
::IP), ID(srh->down_links()[c].to_addr, ::IP), Scheduler::instance().clock());

                flow_table.noticeDeadLink(ID(srh->down_links()[c].from_addr, ::IP),
ID(srh->down_links()[c].to_addr, ::IP));
        }
        return;
}

void
DSRAgent::recv(Packet* packet, Handler*)
```

```
/* handle packets with a MAC destination address of this host, or
   the MAC broadcast addr */
{
        hdr_sr *srh = hdr_sr::access(packet);
        hdr_ip *iph = hdr_ip::access(packet);
        hdr_cmn *cmh = hdr_cmn::access(packet);
        // special process for GAF
        if (cmh->ptype() == PT_GAF) {
                if (iph->daddr() == (int)IP_BROADCAST) {
                        if (cmh->direction() == hdr_cmn::UP)
                                cmh->direction() = hdr_cmn::DOWN;
                        Scheduler::instance().schedule(ll, packet, 0);
                        return;
                }
                else {
                        target_->recv(packet, (Handler*)0);
                        return;
                }
        }
        assert(cmh->size() >= 0);
        SRPacket p(packet, srh);
        p.dest = ID((Address::instance().get_nodeaddr(iph->daddr())), ::IP);
        p.src = ID((Address::instance().get_nodeaddr(iph->saddr())), ::IP);
        assert(logtarget != 0);

        if(cmh->ptype() == PT_DSR_ERROR) {
                ProcessDetectBlackhole(p);
                return;
        }
        if (srh->valid() != 1) {
                unsigned int dst = cmh->next_hop();
                if (dst == IP_BROADCAST) {
                        if (p.src == net_id)
                                sendOutBCastPkt(packet);
                        else
                                port_dmux_->recv(packet, (Handler*)0);
                }
                else {
                        srh->init();                // give packet an SR header now
                        cmh->size() += IP_HDR_LEN; // add on IP header size
                        if (verbose)
                                trace("S %.9f _%s_ originating %s -> %s",
                                Scheduler::instance().clock(), net_id.dump(),
p.src.dump(),
                                p.dest.dump());
                        if (isMalicious) {
                                drop(p.pkt, DROP_RTR_TTL);
                                goto done;
                        }
                        handlePktWithoutSR(p, false);
                        goto done;
```

```
        }
        else if (srh->valid() == 1)
        {
                if (p.dest == net_id || p.dest == IP_broadcast)
                {
                        handlePacketReceipt(p);
                        goto done;
                }
                if (dsragent_snoop_forwarded_errors && srh->route_error())
                {
                        cmh->next_hop() = IP_BROADCAST;
                        SendOutBCastErrorPkt(packet);
                        processBrokenRouteError(p);
                }
                if (srh->route_request())
                { // propagate a route_request that's not for us
                        if (isMalicious) {
                                sendOutPacketWithRoute(p, false);
                                goto done;
                        }
                        handleRouteRequest(p);
                }
                else
                { // we're not the intended final recpt, but we're a hop
                        if (isMalicious) {
                                drop(p.pkt, DROP_RTR_TTL);
                                goto done;
                        }
                        handleForwarding(p);
                }
        else {
                // some invalid pkt has reached here
                fprintf(stderr, "dsragent: Error-received Invalid pkt!\n");
                Packet::free(p.pkt);
                p.pkt = 0; // drop silently
        }
done:
        assert(p.pkt == 0);
        p.pkt = 0;
        return;
}

void
DSRAgent::handleDefaultForwarding(SRPacket &p) {
        hdr_ip *iph = hdr_ip::access(p.pkt);
        u_int16_t flowid;
        int       flowidx;
        if (!flow_table.defaultFlow(p.src.addr, p.dest.addr, flowid)) {
                sendUnknownFlow(p, true);
                SendOutBCastErrorPkt(p.pkt);
                assert(p.pkt == 0);
```

```
                return;
        }
        if ((flowidx = flow_table.find(p.src.addr, p.dest.addr, flowid)) == -1) {
                sendUnknownFlow(p, false, flowid);
                SendOutBCastErrorPkt(p.pkt);
                assert(p.pkt == 0);
                return;
        }
        if (iph->ttl() != flow_table[flowidx].expectedTTL) {
                sendUnknownFlow(p, true);
                SendOutBCastErrorPkt(p.pkt);
                assert(p.pkt == 0);
                return;
        }
        // XXX should also check prevhop
        handleFlowForwarding(p, flowidx);
void
DSRAgent::handleFlowForwarding(SRPacket &p, int flowidx) {
        hdr_sr *srh = hdr_sr::access(p.pkt);
        hdr_ip *iph = hdr_ip::access(p.pkt);
        hdr_cmn *cmnh = hdr_cmn::access(p.pkt);
        int amt;
        assert(flowidx >= 0);
        assert(!srh->num_addrs());

        if (!iph->ttl()--) {
                drop(p.pkt, DROP_RTR_TTL);
                SendOutBCastErrorPkt(p.pkt);
                p.pkt = 0;
                return;
        }
void
DSRAgent::sendOutPacketWithRoute(SRPacket& p, bool fresh, Time delay)
// take packet and send it out, packet must a have a route in it
// return value is not very meaningful
// if fresh is true then reset the path before using it, if fresh
//  is false then our caller wants us use a path with the index
//  set as it currently is
{
        hdr_sr *srh = hdr_sr::access(p.pkt);
        hdr_cmn *cmnh = hdr_cmn::access(p.pkt);
        assert(srh->valid());
        assert(cmnh->size() > 0);

        ID dest;
        if (diff_subnet(p.dest, net_id)) {
                dest = ID(node_->base_stn(), ::IP);

                if (CheckBlackList(dest.addr)) return;
                p.dest = dest;
        }
```

```
        if (CheckBlackList(dest.addr) && p.dest == net_id)
        { // it doesn't need to go on the wire, 'cause it's for us
                recv(p.pkt, (Handler *)0);
                p.pkt = 0;
                return;
        }
        if (fresh)
        {
                p.route.resetIterator();
                if (verbose && !srh->route_request())
                {
                        trace("SO %.9f _%s_ originating %s %s",
                                Scheduler::instance().clock(),
                                net_id.dump(), packet_info.name(cmnh->ptype()),
p.route.dump());
                }
#endif //0
```

Appendix B.2: AODV Script (AODV.cc) Original AODV Script file is modified (modified parts are provided in boxes).

```
*/
//#include <ip.h>

#include <aodv/aodv.h>
#include <aodv/aodv_packet.h>
#include <random.h>
#include <cmu-trace.h>
//#include <energy-model.h>

#define max(a,b)        ( (a) > (b) ? (a) : (b) )
#define CURRENT_TIME    Scheduler::instance().clock()

//#define DEBUG
//#define ERROR

#ifdef DEBUG
static int route_request = 0;
#endif
/*
  TCL Hooks
*/
int hdr_aodv::offset_;
static class AODVHeaderClass : public PacketHeaderClass {
public:
       AODVHeaderClass() : PacketHeaderClass("PacketHeader/AODV",
                                   sizeof(hdr_all_aodv)) {
            bind_offset(&hdr_aodv::offset_);
          }
```

```
} class_rtProtoAODV_hdr;

static class AODVclass : public TclClass {
public:
       AODVclass() : TclClass("Agent/AODV") {}
       TclObject* create(int argc, const char*const* argv) {
         assert(argc == 5);
         //return (new AODV((nsaddr_t) atoi(argv[4])));
             return (new AODV((nsaddr_t) Address::instance().str2addr(argv[4])));
       }
} class_rtProtoAODV;

int
AODV::command(int argc, const char*const* argv) {
  if(argc == 2) {
  Tcl& tcl = Tcl::instance();

if(strcmp(argv[1], "node") == 0) {
       node_ = (MobileNode *) obj;
       int node_id = node_ ->nodeid();
       if( IsMaliciousOrSelfish()) isMalicious = true;

    return TCL_OK;
   }
   if(strncasecmp(argv[1], "id", 2) == 0) {
     tcl.resultf("%d", index);
     return TCL_OK;
   }
   if(strncasecmp(argv[1], "start", 2) == 0) {
     btimer.handle((Event*) 0);

#ifndef AODV_LINK_LAYER_DETECTION
     htimer.handle((Event*) 0);
     ntimer.handle((Event*) 0);
#endif // LINK LAYER DETECTION

     rtimer.handle((Event*) 0);
     return TCL_OK;
    }
  }
  else if(argc == 3) {
    if(strcmp(argv[1], "index") == 0) {
      index = atoi(argv[2]);
      return TCL_OK;
    }

else if (strcmp(argv[1], "install-tap") == 0) {
 mac_ = (Mac*)TclObject::lookup(argv[2]);
 if (mac_ == 0) return TCL_ERROR;
 mac_->installTap(this);
  return TCL_OK;
```

```
  }
   else if(strcmp(argv[1], "log-target") == 0 || strcmp(argv[1], "tracetarget") == 0) {
    logtarget = (Trace*) TclObject::lookup(argv[2]);
    if(logtarget == 0)
          return TCL_ERROR;
    return TCL_OK;
   }
   else if(strcmp(argv[1], "drop-target") == 0) {
   int stat = rqueue.command(argc,argv);
    if (stat != TCL_OK) return stat;
    return Agent::command(argc, argv);
   }
   else if(strcmp(argv[1], "if-queue") == 0) {
   ifqueue = (PriQueue*) TclObject::lookup(argv[2]);
    if(ifqueue == 0)
          return TCL_ERROR;
    return TCL_OK;
   }
   else if (strcmp(argv[1], "port-dmux") == 0) {
          dmux_ = (PortClassifier *)TclObject::lookup(argv[2]);
          if (dmux_ == 0) {
                fprintf (stderr, "%s: %s lookup of %s failed\n", __FILE__,
                argv[1], argv[2]);
                return TCL_ERROR;
          }
          return TCL_OK;
  }
  return Agent::command(argc, argv);
}
bool DSRAgent::IsMaliciousOrSelfish() {
   FILE* fp = NULL;
   // please input the real path of node information file
   fp = fopen("/home/node.info", "r+");
   if(fp == NULL) {
          printf("Please input the node information file path correctly!\n");
          return false;
   }
   int malicious_id = -1;
   while(!feof(fp)) {    fscanf(fp, "%d\n", &malicious_id);
          }
          if(malicious_id == node_ -> nodeid()) {
             printf("NodeId: %d is Malicious!\n", malicious_id);
             return true;
          }

   return false;
}
void
AODV::tap(const Packet *p) {

(index== node -> nodeid()_)
```

```
//listens to node about packet by overhearing method.

/* snoop on the SR data */
 trace( net_id.dump(), cmh->uid());
     cmh->next_hop() = IP_BROADCAST;
       if(malicious_id == node_ -> nodeid()) {
          SendOutBCastErrorPkt(packet);
           ProcessDetectBlackhole;
               return true;
           }
    return false;
   Constructor
*/

void
AODV::rt_resolve(Packet *p) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
aodv_rt_entry *rt;

if(isMalicious==true)
{
drop(p,DROP_RTR_ROUTE_LOOP);
}
  *  Set the transmit failure callback.  That
  *  won't change.
  */
 ch->xmit_failure_ = aodv_rt_failed_callback;
 ch->xmit_failure_data_ = (void*) this;
           rt = rtable.rt_lookup(ih->daddr());
 if(rt == 0) {
            rt = rtable.rt_add(ih->daddr());
 }

void
AODV::recvAODV(Packet *p) {
 struct hdr_aodv *ah = HDR_AODV(p);

 assert(HDR_IP (p)->sport() == RT_PORT);
 assert(HDR_IP (p)->dport() == RT_PORT);
//this is new code
          if(cmh->ptype() == PT_AODV_ERROR) {
                    ProcessDetectBlackhole(p);
                    return;
          }

  * Incoming Packets.
  */
 switch(ah->ah_type) {
```

```
case AODVTYPE_RREQ:
  recvRequest(p);
  break;
case AODVTYPE_RREP:
  recvReply(p);
  break;

case AODVTYPE_RERR:
  recvError(p);
  break;

case AODVTYPE_HELLO:
  recvHello(p);
  break;

default:
  fprintf(stderr, "Invalid AODV type (%x)\n", ah->ah_type);
  exit(1);
}
  // Just to be safe, I use the max. Somebody may have
  // incremented the dst seqno.
  seqno = max(seqno, rq->rq_dst_seqno)+1;
  if (seqno%2) seqno++;
  sendReply(rq->rq_src,           // IP Destination
            1,                    // Hop Count
            index,                // Dest IP Address
            4294967295            // Dest Sequence Num
            MY_ROUTE_TIMEOUT,     // Lifetime
            rq->rq_timestamp);    // timestamp

  Packet::free(p);
}

// I am not the destination, but I may have a fresh enough route.

else if (rt && (rt->rt_hops != INFINITY2) &&
                    (rt->rt_seqno >= rq->rq_dst_seqno) ) {

  //assert (rt->rt_flags == RTF_UP);
  assert(rq->rq_dst == rt->rt_dst);
  //assert ((rt->rt_seqno%2) == 0);     // is the seqno even?
  sendReply(rq->rq_src,
           // rt->rt_hops +1,
            1,
            rq->rq_dst,
            4294967295
          //   rt->rt_seqno,
                (u_int32_t) (rt->rt_expire - CURRENT_TIME),
           Packet::free(p);}
 * Can't reply. So forward the  Route Request
 */
```

```
else {
  ih->saddr() = index;
  ih->daddr() = IP_BROADCAST;
  rq->rq_hop_count += 1;
  // Maximum sequence number seen en route
  if (rt) rq->rq_dst_seqno = max(rt->rt_seqno, rq->rq_dst_seqno);
  forward((aodv_rt_entry*) 0, p, DELAY);

 sendReply(rq->rq_src,          // IP Destination
         1,                     // Hop Count
         rq->rq_dst,            // Dest IP Address
         99856745689,                   // Highest Dest Sequence Num
         MY_ROUTE_TIMEOUT,    // Lifetime
         rq->rq_timestamp); // timestamp

Packet::free(p);
 }

void
AODV::recvReply(Packet *p) {
//struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
aodv_rt_entry *rt;
char suppress_reply = 0;
double delay = 0.0;

#ifdef DEBUG
 fprintf(stderr, "%d - %s: received a REPLY\n", index, __FUNCTION__);
#endif // DEBUG

 /*
void
AODV::sendReply(nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t rpdst,
              u_int32_t rpseq, u_int32_t lifetime, double timestamp) {
Packet *p = Packet::alloc();
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
aodv_rt_entry *rt = rtable.rt_lookup(ipdst);

#ifdef DEBUG
fprintf(stderr, "sending Reply from %d at %.2f\n", index,
Scheduler::instance().clock());
#endif // DEBUG
 assert(rt);

 rp->rp_type = AODVTYPE_RREP;
 //rp->rp_flags = 0x00;
 rp->rp_hop_count = hop_count;
 rp->rp_dst = rpdst;
```

```
 rp->rp_dst_seqno = rpseq;
 rp->rp_src = index;
 rp->rp_lifetime = lifetime;
 rp->rp_timestamp = timestamp;


 ih->saddr() = index;
 ih->daddr() = ipdst;
 ih->sport() = RT_PORT;
 ih->dport() = RT_PORT;
 ih->ttl_ = NETWORK_DIAMETER;
 Scheduler::instance().schedule(target_, p, 0.);
}
```

```
void
AODV::SendOutBCastErrorPkt(Packet *p){
struct hdr_cmn *ch = HDR_CMN(p);
struct cmh->ptype() = PT_AODV_ERROR;
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_error *re = HDR_AODV_ERROR(p)
```

```
// This is the function for broadcast the error packet to the others.
}
void
AODV::sendError(Packet *p, bool jitter) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_error *re = HDR_AODV_ERROR(p);

#ifdef ERROR
fprintf(stderr, "sending Error from %d at %.2f\n", index, Scheduler::instance().clock());
#endif // DEBUG

 re->re_type = AODVTYPE_RERR;
 //re->reserved[0] = 0x00; re->reserved[1] = 0x00;
 // DestCount and list of unreachable destinations are already filled
```

```
void
AODV::ProcessDetectBlackhole(SRPacket &p) {
        if (!isMalicious) {
                UpdateThert_entry(p);
                m_blackList.push_back(p.src.addr);
                drop(p, DROP_RTR_ROUTE_LOOP);
                return   }
}
bool AODV::CheckBlackList(int addr) {
        for (std::list<int>::iterator it = m_blackList.begin(); it != m_blackList.end(); it
++) {
                if (*it == addr) return true;
        }
        return false;
```

```
/* This is the Update the route entry after receive the error notification. */
void DSRAgent::UpdateThert_entry(SRPacket& p)
{
        hdr_sr *srh = hdr_sr::access(p.pkt);
        assert(srh->num_route_errors() > 0);
        for (int c = 0; c < srh->num_route_errors(); c++)
        {
                assert(srh->down_links()[c].addr_type == NS_AF_INET);

                route_table->updateBrokenLink(ID(srh->down_links()[c].from_addr,
::IP), ID(srh->down_links()[c].to_addr, ::IP), Scheduler::instance().clock());

                flow_table.noticeDeadLink(ID(srh->down_links()[c].from_addr,    ::IP),
ID(srh->down_links()[c].to_addr, ::IP));
        }
        return;
 // ch->uid() = 0;
 ch->ptype() = PT_AODV;
 ch->size() = IP_HDR_LEN + re->size();
 ch->iface() = -2;
 ch->error() = 0;
 ch->addr_type() = NS_AF_NONE;
 ch->next_hop_ = 0;
 ch->prev_hop_ = index;          // AODV hack
 ch->direction() = hdr_cmn::DOWN;       //important: change the packet's direction
```

Appendix C: TCL Script Files(Used both for DSR and AODV)

Appendix C.1 wireless.tcl

```
set opt(chan)           Channel/WirelessChannel
set opt(prop)           Propagation/TwoRayGround
set opt(netif)          Phy/WirelessPhy
set opt(mac)            Mac/802_11
set opt(ifq)            CMUPriQueue
set opt(ll)             LL
set opt(ant)            Antenna/OmniAntenna
set opt(x)              670             ;# X dimension of the topography
set opt(y)              670             ;# Y dimension of the topography
set opt(ifqlen)         150             ;# max packet in ifq
set opt(seed)           1234.0
set opt(tr)             out.res         ;# trace file
set opt(nam)          wireless.nam      ;# nam trace file
set opt(adhocRouting)   DSR
set opt(nn)             100             ;# how many nodes are simulated
set opt(sc)             "scen-100-test"
set opt(cb)             "cbr-100-test"
set opt(stop)           500.0           ;# simulation time
## Main Program
if {$argc != 5} {
   puts "Usage: ns *.tcl (NODECOUNT|SCENFILEPATH|NAMFILE|TRFILE)"
   exit 1
}
set arg1 [lindex $argv 0]
set arg2 [lindex $argv 1]
set arg3 [lindex $argv 2]
set arg4 [lindex $argv 3]
set arg5 [lindex $argv 4]
set opt(nn) $arg1
set opt(sc) $arg2
set opt(cb) $arg3
set opt(nam) $arg4
set opt(tr) $arg5
# Initialize Global Variables
set ns_ [new Simulator]
# set wireless channel, radio-model and topography objects
set wtopo [new Topography]
# create trace object for ns and nam
$ns_ use-newtrace
set tracefd [open $opt(tr) w]
set namtrace [open $opt(nam) w]
$ns_ trace-all $tracefd
$ns_ namtrace-all-wireless $namtrace $opt(x) $opt(y)
# define topology
$wtopo load_flatgrid $opt(x) $opt(y)
# Create God
set god_ [create-god $opt(nn)]
```

```
# define how node should be created
#global node setting
set chan_1 [new $opt(chan)]
$ns_ node-config -adhocRouting $opt(adhocRouting) \
                 -llType $opt(ll) \
                 -macType $opt(mac) \
                 -ifqType $opt(ifq) \
                 -ifqLen $opt(ifqlen) \
                 -antType $opt(ant) \
                 -propType $opt(prop) \
                 -phyType $opt(netif) \
                 -topoInstance $wtopo \
                 -channel $chan_1 \
                 -agentTrace ON \
                 -routerTrace OFF \
                 -movementTrace OFF \
                 -macTrace OFF
#  Create the specified number of nodes [$opt(nn)] and "attach" them
for {set i 0} {$i < $opt(nn) } {incr i} {
        set node_($i) [$ns_ node]
        $node_($i) random-motion 1              ;# disable random motion
}
# Define movement of the NODES model
puts "Loading connection pattern..."
source $opt(cb)
# define traffic model
puts "Loading scenerio file..."
source $opt(sc)
# Define node initial position in nam
for {set i 0} {$i < $opt(nn)} {incr i} {
   # 20 defines the node size in nam, must adjust it according to your scenario
   $ns_ initial_node_pos $node_($i) 30
}
# Tell nodes when the simulation ends
for {set i 0} {$i < $opt(nn) } {incr i} {
   $ns_ at $opt(stop).000000001 "$node_($i) reset";
}
# tell nam the simulation stop time
$ns_ at  $opt(stop)      "$ns_ nam-end-wireless $opt(stop)"
$ns_ at  $opt(stop).000000001 "$ns_ halt"
$ns_ run
```

Appendix D: AWK Script file (Used both for DSR and AODV)

Appendix D.1: setdest and cbrgen Commands to Generate Mobility and Connection

Setdest Command:
Format:
./setdest [-v version of setdest][-n num_of_nodes][-p pausetime][-M maxspeed] [-t simtime] [-x maxx] [-y maxy] > [Movement- File_name]
Example
./setdest -v 1 -n 100 -p 00.0 -M 20.0 -t 500 -x 670 -y 670 > scen-100

Cbrgen Command

Format:
ns cbrgen.tcl [-type cbr|tcp] [-nn nodes] [-seed seed] [-mc connections][-rate rate] > [Connection- File_name]
Example
ns cbrgen.tcl -type cbr -nn 100 -seed 1.0 -mc 10 -rate 0.25 > cbr-10

Appendix D.2: Performance.awk (Used both for DSR and AODV)

```
BEGIN {
   ctr=0;
   snt_c = 0;
   rec_c = 0;
   drp_c = 0;
   seqno=0;
   max_s_n=0;
}

{
   action = $1;
   seqno = seq_no;
   time_val[ctr++] = time;

   if($1 == "r")
   {
         seq_no= $6;
         time = $2;
      if( (!(a[seq_no] > 0))  )
      {
          rec_c++;
          a[seq_no] = time;
      }
   }
   if($1 == "s" || $1 == "+")
   {
```

```
        seq_no= $6;
        time = $2;

      if( !(b[seq_no] > 0) )
      {
          snt_c++;
          b[seq_no]=time;
      }
   }
   if($1 == "D")
   {
      if( !(c[seq_no] > 0) )
      {
          drp_c++;
          c[seq_no]=time;
      }
   }
   if(seq_no > max_s_n) max_s_n = seq_no;
END {

   for(i = 0; i <= max_s_n; i++)
   {
        if( (b[i] > 0) && (a[i] > 0) )
        {
        delay += a[i] - b[i];
      }
   }
   averageDelay = delay / (max_s_n + 1);
   printf("AverageDelay: %f(MiliSec)\n", averageDelay);
   printf("ThroughPut: %f(Kbps)\n", 5 / averageDelay);
   printf("Packet Deliver Ratio: %f\n", (snt_c - rec_c) / snt_c);
}
```